CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Facultad de Informática

TRABAJO FIN DE GRADO

Editor visual de programas y mapas para RoboMind

Autor: Diego Alberto Vidal

Director: Manuel Collado

MADRID, JUNIO DE 2013

What a computer is to me is it's the most remarkable tool that we've ever come up with, and it's the equivalent of a bicycle for our minds.

Steve Jobs

Para mi amor.

RESUMEN

RoboMind es un programa educativo utilizado en todos los niveles educativos, desde el colegio hasta la universidad. Este programa simula un robot que se desplaza a través de un mapa. Este proyecto surge de la necesidad de ampliar ciertas funcionalidades de dicho programa.

Para la realización del mismo se han utilizado las tecnologías proporcionadas por *Java,* utilizando como base el código fuente de libre distribución.

Este proyecto cuenta con partes de diseño y partes de implementación, en la que se ha utilizado metodologías orientadas a objetos.

ABSTRACT

RoboMind is an educational programming environment used in all academic disciplines from primary school to college. This application simulates a robot that can move around a world. This project comes from the necessity of extending certain functionalities of it.

The technologies used for developing has been those provided by the *Java* framework, using the free program sources as support for the project.

The project has two parts, one design part and another, implementation part, in which object oriented technologies had been used.

AGRADECIMIENTOS

Quiero agradecer desde aquí a todas aquellas personas que han estado conmigo durante tantos años de estudio en esta universidad y que me han apoyado en todas las situaciones, tanto las buenas, como las no tan buenas. Han sido unos años maravillosos, lleno de buenos recuerdos y momentos, recuerdos que me acompañarán durante toda mi vida.

Muchos de estos compañeros no sólo han estado allí siempre, sino también me han ayudado a hacer posible mi carrera.

Quisiera agradecer específicamente a mis compañeros de siempre, compañeros que al final se han convertido en amigos y algunos en confidentes: Héctor, Paola, Sergio, Álvaro, Roberto, Iñaqui, Danieles (varios), Vila y un largo etcétera. También a aquellos que no siendo compañeros hemos compartido algún hueco en la facultad, y que tanto hemos debatido sobre el tema que más nos interesa de todos: la informática.

Quisiera hacer una mención especial a Guadalupe, que gracias a su perfección lingüística ha hecho de este proyecto un poema de corrección sintáctica y gramatical. Gracias por leerte este *tostón* que poco tiene que ver contigo.

Por último quería agradecer a la persona más importante, a Miguel. Creo que todo cobra sentido cuando estoy a tu lado. Gracias por aguantar tantos años de estudio a mi lado, facilitándome las cosas, estando ahí cuando te necesitaba.

A todos, muchas gracias.

Diego Alberto Vidal

Junio 2013

Índice

1. INTRODUCCIÓN

1.1. Introducción al TFG

El presente trabajo de fin de grado pretende añadir funcionalidades al entorno de programación *RoboMind.* Las funcionalidades que se le desean añadir son: el editor visual de mapas; el editor visual de programas.

RoboMind es un sistema para enseñanza de programación a todos los niveles (primaria, secundaria, bachillerato o universidad). Se basa en un robot simulado que se mueve sobre un mundo virtual (o "mapa") en forma de plano cuadriculado con obstáculos y elementos móviles. El robot se programa mediante un lenguaje de programación (llamado Robo) relativamente sencillo e intuitivo. Robomind es un descendiente de otros sistemas anteriores, tales como Karel o Logo.

Para realizar las funcionalidades se utilizará el código fuente de dicho programa en su versión de libre distribución, se estudiará a fondo y se desarrollará e integrarán al programa actual las funcionalidades de edición visual de mapas y de programas.

Esta tarea puede resultar ardua, ya que implica estudiar un código ajeno y complejo. Esta tarea se dificulta aún más si el código en cuestión no está comentado en absoluto y además no sigue alguna pauta de diseño como puede ser el patrón de diseño Modelo-Vista-Controlador (MVC).

En la figura 1.1 se puede observar el entorno del actual programa, al cual se le pretenden añadir las distintas funcionalidades. El programa utiliza las librerías gráficas de Java de *Swing* para el desarrollo de la interfaz. Por lo tanto, es

estrictamente necesario el profundo conocimiento de esta librería, así como de otras que utiliza el programa y que se detallarán más adelante en este proyecto.

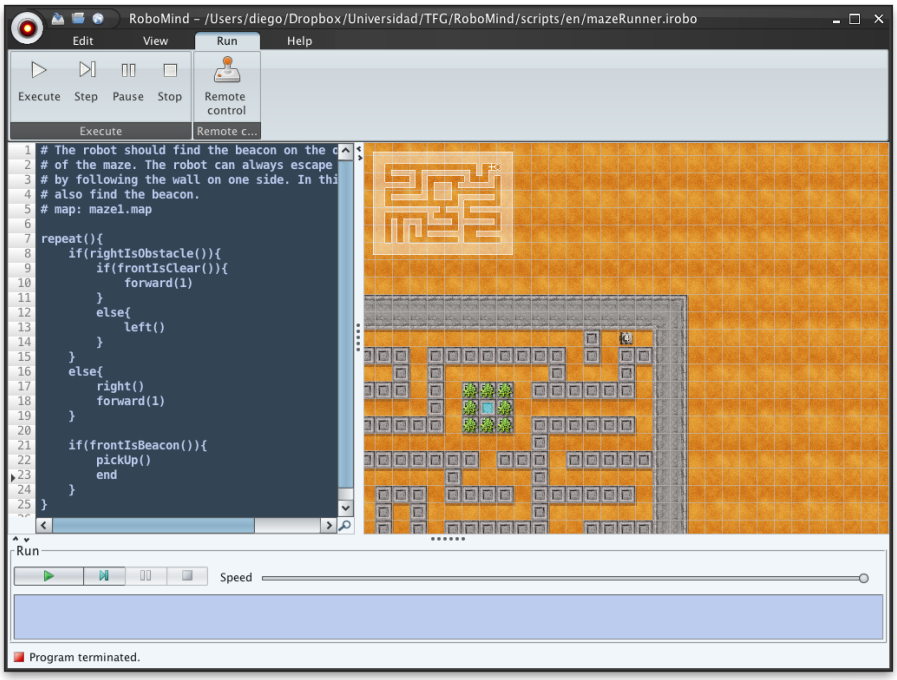

Figura 1.1. *Entorno de programación de RoboMind*

Como se puede observar, el programa consta de dos paneles, uno con el mapa y otro con el código. Se pretende poder modificar tanto el código como el mapa de una manera visual e interactiva, facilitando al usuario tanto la tarea de codificación como la de creación de mapas, la cual sólo era posible hasta ahora, mediante el uso de un editor de textos externo y empleando una sintaxis proporcionada por el programa.

1.2. Motivación

La motivación principal de desarrollar este proyecto viene dada por mi interés dentro del campo del desarrollo de aplicaciones Java. Desde pequeño siempre he estado interesado en la programación (aprendí a programar en C con sólo 10 años

por mi cuenta), y desde que comencé mi carrera en este centro, me he interesado por muchos lenguaje. Sin embargo, ninguno me llamó tanto la atención como Java. Desde entonces he estudiado muchísimo este lenguaje, explorando todas las posibilidades que ofrecía con sus numerosas librerías. Por lo que este proyecto ha suscitado mi interés desde el principio.

1.3. Punto de partida

Como comentaba anteriormente, este proyecto pretende añadir funcionalidades a un entorno de programación, así que, el punto de partida sería obtener el código fuente de dicha aplicación. En un principio, este programa se distribuía con código abierto, pero ya no se distribuye de esta manera en sus versiones más recientes. Por lo que ha sido necesario obtener la última versión de la cuál se dispone de código fuente para poder comenzar a desarrollar el proyecto.

Como entorno de desarrollo Java, se utilizará NetBeans, en su versión 7.3 y la última versión de Java, la 6.

1.4. Objetivos

Se exponen a continuación la lista con los principales objetivos del proyecto:

- Construir un editor visual de mapas usando los mismos elementos gráficos que el entorno RoboMind.
- Integrar dicho editor de mapas en el entorno RoboMind.
- Idear una representación gráfica de los elementos estructurales del lenguaje Robo similar a la de Scratch.
- Construir un editor visual de programas en Robo, usando la representación anterior.

- Integrar dicho editor de programas en el entorno RoboMind.
- Validar las nuevas herramientas reproduciendo los ejemplos ya incuidos en el entorno Robomind y/o creando otros nuevos.

1.5. Estructura del proyecto

El proyecto durante su desarrollo, se ha dividido en distintas fases. Como se ha podido observar, tenemos que realizar un estudio a fondo del código fuente para su comprensión y para determinar qué funcionalidad tiene cada elemento del mismo y, de esta manera, facilitar las tareas de integración, ya que las funcionalidades deberían estar embebidas en el programa, es decir, no tendrían que parecer externas. Por lo cual, parece natural que la primera parte del proyecto sea el estudio del código, la documentación y la recopilación de información acerca de librerías, etc. Tanto los detalles del programa, como el estudio del código fuente se realizará a través de los capítulos 2 y 3 del presente proyecto.

Una vez realizado esto, ya se podrá determinar qué parte desarrollar primero: por un lado el editor gráfico de mapas y, por otro, el editor gráfico de programas.

La parte de menor complejidad de este proyecto es, sin lugar a dudas, el editor gráfico de mapas. Por esta razón, resulta evidente que se abordará éste en primer lugar. En el capítulo 4, se explicará el desarrollo de dicha funcionalidad, así como los aspectos más destacados y los cambios más relevantes realizados en el código original del programa para integrarla.

La parte del editor de programas tiene una mayor complejidad. En primer lugar porque es necesario elaborar una manera de representar gráficamente un programa. También añade una dificultad extra el tener que poder pasar de un fichero de texto que contenga el programa a la representación gráfica y viceversa.

Como se explicará en la sección 5 este objetivo no ha sido posible abordarlo debido a cómo está representados internamente los programas. En cualquier caso se propondrá una posible solución para abordar este problema, aunque ésta se salga del alcance del presente proyecto.

Por último, el capítulo 6 estará dedicado a las conclusiones de este proyecto y a las posibles líneas de desarrollo para realizar más mejoras en este entorno y, sobre todo, en lo expuesto en este trabajo.

2. ESTADO DEL ARTE

2.1. RoboMind

El objetivo del Trabajo de Fin de Grado es el desarrollo de una extensión para el programa de enseñanza de programación RoboMind. Esta extensión se basa en el desarrollo de dos editores gráficos, uno de mapas y otro de programas.

RoboMind es un programa educativo que se utiliza en todos los niveles de educación, desde primaria hasta bachillerato e, incluso, en universidad. El objetivo de este programa es la enseñanza de conceptos básicos de programación mediante la programación de un robot simulado.

Utilizando un lenguaje de programación llamado Robo, el cual tiene muchas similitudes con C, el usuario puede dar órdenes al robot, para que éste se mueva a través de un mundo que es una cuadrícula bidimensional, en la cual puede haber otros elementos como obstáculos u objetos que pueden ser apresados por el robot.

Este lenguaje es un lenguaje de programación de alto nivel básico, consistente en una serie de grupos de funciones, estás son:

1. **Funciones de movimiento del robot**: *forward(n), backward(n),left(), right(), north(n), south(n), west(n), east(n)* para mover el robot hacia delante, hacia atrás, girarlo a la izquierda, girarlo a la derecha, avanzar hacia el norte, avanzar hacia el sur, avanzar hacia el oeste y hacia el este respectivamente. El argumento *n* de algunas funciones, determina el número de pasos que va a dar el robot.

2. **Funciones que permiten "ver" el medio**: *frontIsClear(), frontIsObstacle(), frontIsBeacon(), frontIsWhite(), frontIsWhite()* son funciones booleanas que devuelven si enfrente del robot está libre, hay un obstáculo, hay una baliza, es un camino blanco o es un camino negro respectivamente. De igual manera cambiando *front* por *left* hacemos lo mismo pero para el lado izquierdo del robot (*leftIsClear(), leftIsObstacle(), ...)* y cambiando *front* por *right* "miramos" el lado derecho del robot (*rightIsClear(), rightIsObstacle(), ...).*

3. **Funciones de pintar**: el lenguaje nos permite hacer que el robot vaya dejando un rastro de por dónde ha ido. De esta manera tenemos *paintWhite(), paintBlack()* y *stopPainting()* que inicia a pintar en blanco o en negro o que para de pintar.
4. **Recogida de balizas**: el robot puede recoger una baliza o bien dejarla (si la lleva consigo) utilizando las funciones *pickUp()* y *putDown()* respectivamente.

5. **Condicionales**: el programa tiene cambios de flujo condicionales mediante las estructuras *if, if-else, if-else-if-else…*

6. **Bucles**: los bucles se realizan mediante *repeat()* que sin argumentos repite siempre, y con argumento un número, el número de veces que se repite su contenido. También tiene *repeatWhile(contidion)* que repite mientras se cumpla la condición.

7. **Definición de procedimientos**: el lenguaje permite definir procedimientos mediante *procedure p { }*. Permite salir de la función mediante *return.*

8. **Tirar una moneda**: *flipCoin()* retorna un valor aleatorio bien verdadero, bien falso.

9. **Fin del programa**: *end* finaliza la ejecución del programa.

Si bien los comando anteriores están en inglés, el lenguaje está disponible en 22 idiomas, de tal manera que los comando también se pueden escribir en otros idiomas (por ejemplo, podemos escribir *repetir* en vez de *repeat*).

Actualmente, las últimas versiones del programa, permiten que los programas de RoboMind se ejecuten directamente en un robot de verdad. En estos momentos sólo están soportados los robots de Lego Mindstorms NXT 2.0. que son los que se observan en la figura 2.1.1

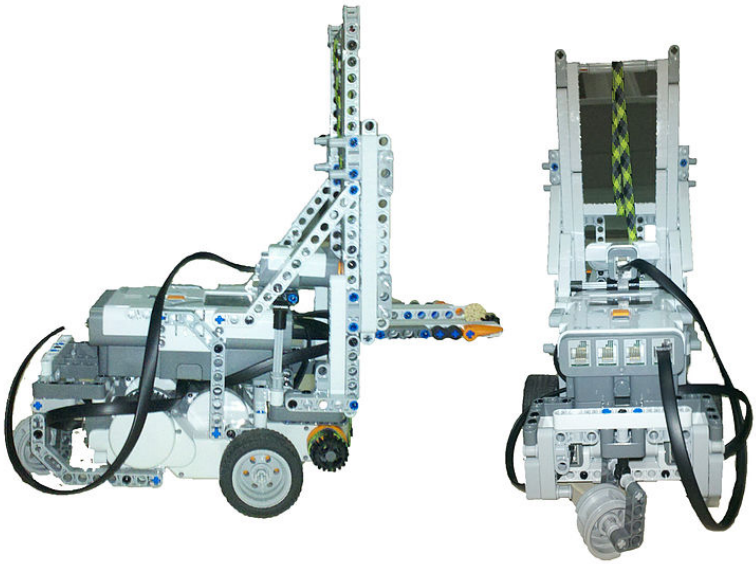

Figura 1.1.1 ***Modelo construido con Lego Mindstorms NXT 2.0.***

2.2. Editores visuales de programas

Los editores visuales de programas son lenguajes que nos permiten crear programas a través de la manipulación de elementos gráficos en vez de mediante la escritura de comandos como es habitual. Estos editores tienen su sintaxis, pero en vez de ser estructuras gramaticales son expresiones visuales y símbolos. Toda esta "sintaxis visual" tendría su equivalente textual, lo cual significa que luego es transformado a un documento de texto, aunque muchas veces las configuraciones que puede adoptar un programa de estas categorías no parecen tener una traducción tan inmediata.

En la actualidad existen multitud de lenguajes visuales, muchos de ellos orientados a la educación. Algunos ejemplos son *Alice*, un lenguaje que utiliza *drag&drop* para crear animaciones utilizando modelos tridimensionales; o el famoso *Scratch*, que también se basan en el *drag&drop* para arrastra una serie de bloques que permiten mover una serie de personajes sobre un fondo seleccionado.

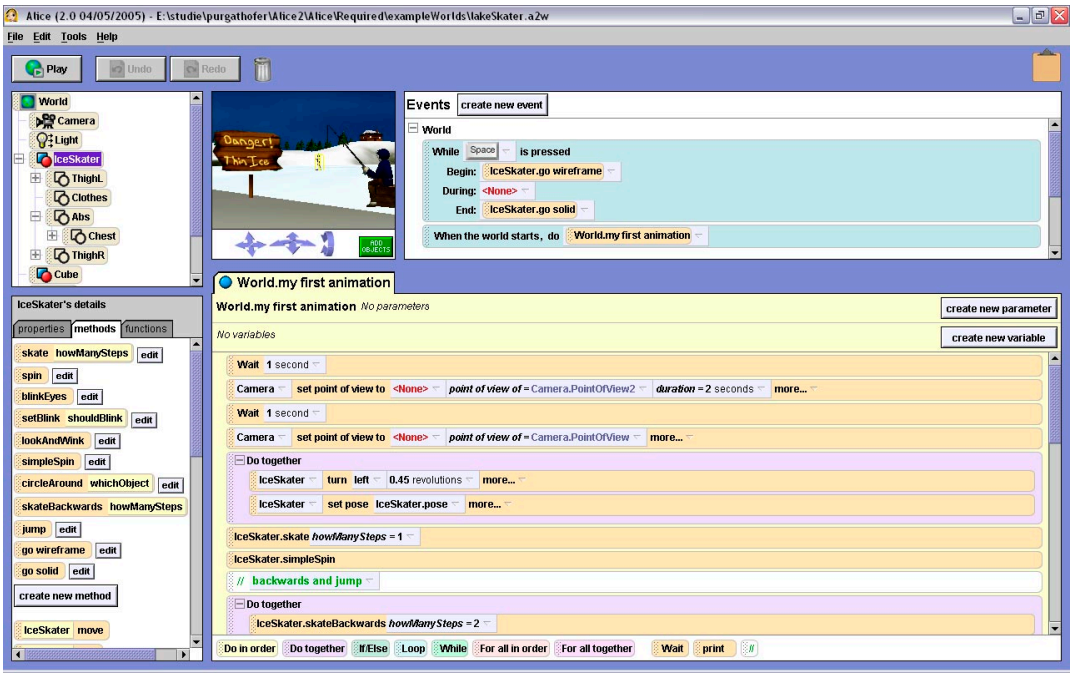

Figura 2.2.1. *Entorno de programación de Alice*

3. CÓDIGO FUENTE DEL PROGRAMA

En esta sección se pretende detallar la estructura del código fuente de RoboMind, haciendo especial hincapié en aquellas clases que son necesarias para el desarrollo de este proyecto.

3.1. Patrones utilizados

RoboMind está desarrollado siguiendo un modelo parecido al Modelo-Vista-Controlador (MVC). El modelo MVC es un patrón de arquitectura de las aplicaciones software, en el cual se separa la lógica del programa de la interfaz del usuario. Esto permite, entre otros: la evolución independiente de ambos aspectos; la posibilidad de reutilización de código; flexibilidad del programa. Por defecto, Java en su librería Swing (librería gráfica para Java que se explicará más adelante) utiliza este patrón permitiendo que toda la aplicación pueda ser desarrollada siguiendo el mismo.

Los componentes de MVC y, especialmente en Swing, son los siguientes:

- **Modelo**: es la representación específica de la información con la cual el sistema opera. Debe gestionar pues, todos los accesos a dicha información, incluyendo lecturas, actualizaciones, e incluso los privilegios de acceso de la aplicación. En Java, son todas las estructuras de datos necesarias para almacenar la información del programa y también los propios datos.

- **Vista**: es la representación gráfica del modelo. Esta representación tiene un formato adecuado para que el usuario pueda interactuar, por tanto requiere de la información del modelo. En Java, son todos los objetos

que heredan de la clase java.awt.Component, y que, por ende, son representables.

- **Controlador**: reacciona a petición del cliente, ejecutando dicha petición y realizando los cambios pertinentes sobre el modelo. En el caso particular de Java, el controlador es el hilo de tratamiento de eventos que captura y propaga los eventos al modelo (y a la vista). Se utilizan clases que implementan las interfaces *ActionListener, MouseListener,* entre otras.

Por tanto, el modelo MVC podría representarse mediante el siguiente gráfico:

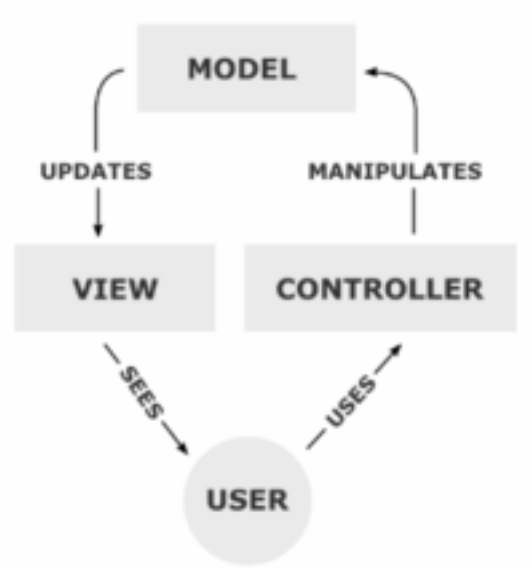

Figura 3.1. *Esquema del patrón MVC*

En el caso de este programa, el controlador es la clase *Controller* del paquete *robo*. Esta clase junta la interfaz del usuario con las acciones. Además sigue el patrón *Singleton*, que tiene como misión garantizar que sólo exista una instancia de dicha clase en todo el programa y proporcionar un punto de acceso global a ella.

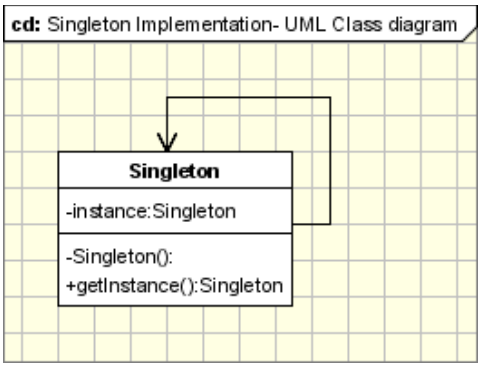

Figura 3.2. *Esquema UML del patrón Singleton.*

También se hace uso del patrón *Prototype* el cual permite crear objetos personalizados sin la necesidad de conocer la clase de los mismo ni como crearlos. En este programa, se ha utilizado dicho patrón para poder implementar diferentes versiones de una interfaz, facilitando así las tareas de modificación o ampliación del programa.

3.2. Controlador

Como se mencionaba en el apartado anterior, el programa sigue una estructura basada en el patrón MVC, con la clase *Controller* como controlador del patrón. Lo cierto es que el patrón MVC en este programa no se cumple con severidad. Se puede notar el esfuerzo por separar cada una de las partes involucradas en el patrón, pero finalmente se mezclan todos los elementos en la clase *Controller*. Además esta clase no sólo controla, sino que crea los elementos de la interfaz gráfica, carga el modelo, etc.

El controlador es instanciado por él mismo cuando la clase principal (*RoboMind*) llama a su método *init(char[] args, boolean useSubstance)*, que es un método estático que crea una nueva instancia de *Controller* si ésta no existe ya.

Al iniciarse, el controlador instancia otras clases fundamentales para el funcionamiento del programa (que se detallarán más adelante) crea las acciones

para responder a la interactuación del usuario y por último crea todos elementos gráficos.

Las acciones de usuario están definidas por una serie de clases anidadas dentro de la clase *Controller*. Estas extienden la clase *RoboAction* que, a su vez, extiende *AbstracAction.* Esta última sirve para subscribirse a eventos de la interfaz del usuario (como por ejemplo que un usuario pulse un botón, o que pase el ratón por determinada zona). Al ocurrir el evento, las clases subscritas reciben el mensaje en forma de invocación del método *actionPerformed* que implementan.

También contiene como atributos referencias a la implementación de la máquina virtual, de la que hablaremos más adelante, y del mundo actual.

A continuación se van a detallar las clases que son instanciadas por Controller así como otras clases importantes para el desarrollo de este proyecto y su cometido en el programa:

3.2.1. ResourceManager

Esta clase es la encargada de administrar todos los recursos del programa. Al instanciarse, carga el fichero de configuración del programa para leer algunas de las configuraciones que necesita. Seguidamente crea unos *ResourceBundle,* leyendo del directorio del programa los ficheros adecuados. Los *ResorceBundle* son un recurso que proporciona la plataforma Java para almacenar objetos relacionados con el *locale* actual del sistema. De esta manera, el programa puede ser traducido a cualquier idioma. Cada vez que se necesite un texto, se busca como si se tratase de un *Map* en el que a una clave le corresponde un valor.

También carga las imágenes necesarias para el programa, así como los distintos *skins* disponibles para pintar los mapas. Toda esta información la va almacenando componiendo así una parte del *Modelo* del programa.

Esta clase está estrechamente relacionada con el *Controller* aunque no sabe de la existencia del mismo, por lo que es independiente.

3.2.2. World

Se trata de otra clase que forma parte del *Modelo* del programa. Representa al robot en su ambiente.

Esta clase contiene todos los métodos necesarios para controlar el robot, y manipularlo en todas las direcciones posibles. Además también permite coger balizas y soltarlas.

3.2.3. TitleMap

Esta clase es un plano que representa paredes, balizas y trazas de pintura. Se trata de la representación del fichero de mapas en el sistema y contiene toda su información: su tamaño, los obstáculos, las decoraciones y las balizas.

Incluye una serie de métodos que permiten saber si hay un obstáculo en algún punto del mapa, actualizar las trazas de pintura, etc. No permite, sin embargo, manipular el mapa, para aumentar sus dimensiones o añadir nuevos obstáculos, lo cual sería muy útil para el objetivo de este proyecto.

Es necesario conocer el tamaño del mapa y los caracteres que lo representan para instanciar esta clase, la cual procesa dichos caracteres y actualiza sus estructuras internas para realizar la representación.

3.3. Interfaz Gráfica

Los elementos de la interfaz gráfica se encuentran en el paquete *robo.gui.* En este podemos encontrar las distintas clases que forman parte no sólo de la interfaz visual, sino también de elementos que controlan la misma. Algunas de estas clases son fundamentales tanto para el programa en sí como para el desarrollo del presente proyecto.

A pesar de tener un paquete específico para la interfaz gráfica, la ventana principal, (*JFrame*), no se encuentra en este paquete, sino que lo crea y puebla el mismo con los elementos necesarios, algunos pertenecientes a este paquete. Aquí es donde se evidencia la falta de modularidad del programa, en algunos aspectos, del programa. Mezclar la creación de elementos visuales con la lógica del programa hace afecta a la escalabilidad del programa, y hace difícil que programadores ajenos al mismo puedan ampliarlo.

A continuación se detallarán las clases más importantes en la interfaz gráfica.

3.3.1. Animation

Una de las clases más utilizadas en el programa es la clase *Animation*, que representa una serie de imágenes y la cantidad de tiempo que se visualiza cada una. Se utiliza para representar los elementos en el mapa. Estos elementos pueden ir cambiando su imagen, es decir, que pueden estar animados. Por ejemplo, el obstáculo agua, da la sensación de que el agua tiene movimiento. Todo lo que se va a visualizar del mapa es de la clase *Animation*, tanto el robot, como los obstáculos, incluso las trazas de pintura. No es necesario que *Animation* contenga varias imágenes, también puede ser una imagen estática.

3.3.2. IMonitor y Monitor

Se trata de una clase abstracta, que extiende *JPanel*, por lo que es un elemento visualizable que, en principio, formaría parte de la "V" en MVC. Esta interfaz es la representación gráfica de un mundo. Controla todas las acciones relacionadas con él, tanto el zoom y el desplazamiento del mundo, por tanto, la zona visible por el usuario, como todos los movimientos y acciones del robot, como sus movimientos, visualizar en las direcciones posibles, coger elementos, etc. Esto nos lleva a la conclusión de que una vez más se mezclan todos los componentes de MVC en una sola clase, ya que ésta, aparte de responder a las acciones del usuario, es también la vista del usuario y hasta hace las veces de modelo como veremos a continuación.

La clase que extiende *IMonitor* es la clase *Monitor,* una de las clases más importantes del programa. Es instanciada por *Controller* en el momento de iniciar la interfaz gráfica y dado que extiende a *JPanel* puede añadirlo directamente al *JFrame* principal del programa.

Monitor es una clase que posee multitud de funcionalidades. Para empezar tiene un conjunto de atributos que conforman el estado interno del mundo representado: el zoom; el desplazamiento; la existencia o no de cuadrícula de referencia; el tamaño de la misma, y muchos otros estados. También contiene información acerca de la posición actual del robot, y una serie de *Animations* que utilizará en caso de tener que realizar alguna acción, por ejemplo, pintar las trazas de pintura, valga la redundancia, o tirar una moneda, que, como se explicó anteriormente, es la representación gráfica de obtener un valor aleatorio.

Para la representación del mundo se utilizan una serie de atributos:

```
38private double scale;
54private double tx, ty;
```

El atributo *scale* indica la escala actual del monitor y los atributos *tx* y *ty* indican su traslación. Además Monitor sustituye el método *paint* de *JPanel* para usar uno propio. Por ende, el tamaño del *JPanel* inherente es fijo, solo que el método *paint* que *Monitor* sobreescribe, a través de los atributos anteriores dibuja el mapa, teniendo en cuenta este estado. Por tanto, cuando el usuario arrastra el ratón, los valores *tx* y *ty* cambian, por lo que *Monitor* sabe qué es lo que tiene que pintar en cada momento. Consecuentemente, y dado que los valores *tx* y *ty* no tienen limitación alguna, tenemos un mapa de tamaño ilimitado.

Cabe destacar que los valores *tx* y *ty* son valores continuos, mientras que los objetos del mapa se encuentran en posiciones discretas de tuplas del tipo (x, y), en la sección del desarrollo del editor de mapas se especificará como resolver este problema.

Por último, hay que especificar que el origen de coordenadas está en la esquina superior izquierda. A partir de ahí, el eje "X" crece de izquierda a derecha, y el eje "Y" crece de arriba abajo.

3.4. Compilador

RoboMind es un lenguaje de alto nivel, el cual necesita ser compilado para posteriormente ser ejecutado. Al examinar detenidamente este proceso observa que el compilador genera directamente una lista de elementos parecidos al *bytecodes.* El *bytecode* es un código intermedio, en el que cada elemento es una operación con un conjunto de parámetros como argumentos. Lo esperado en este punto sería que el programa generara un árbol de sintaxis abstracta (AST) y después procediera a ejecutar el programa siguiendo esta estructura arborescente. Como ventaja sobre el *bytecode,* el AST mantiene la estructura y las relaciones

globales del programa entre las sentencias y proporciona una representación más compacta [15][16].

En cualquier caso, al recibir la orden del usuario de ejecutar el código, el programa primeramente procesa todo el código en busca de errores sintácticos y semánticos, y, seguidamente, se procesan los resultados. Por lo tanto estamos hablando compilador con dos fases. Para la parte sintáctica desarrolla un muy simple compilador descendente recursivo. Básicamente, se resume en las siguientes líneas:

```
1  boolean b = true;
2  while(b){
3      //tryParseLabel();
4
5      // in specific to general order!!
6      b = tryParseBockClose()
7        || tryParseBreak()
8        || tryParseReturn()
9        || tryParseEnd()
10       || tryParseBeginRepeatWhile()
11       || tryParseBeginRepeat()
12       || tryParseBeginIf()
13       || tryParseProcDef()
14       || tryParseCall();
15 }
```

Lo que este algoritmo hace es comprobar si el *token* actual se corresponde con el *first* de alguna de las reglas gramaticales definidas en cada uno de los métodos. Si lo es continúa comprobando el resto de *tokens,* en el caso de cumplir la regla, el método devuelve verdadero, si el fallo se produce en alguno de los pasos intermedios, el método tira una excepción correspondiente a un error sintáctico. Si el *first* no corresponde a ninguno, es que hay un fallo sintáctico. Además, cada método va generando el *bytecode* correspondiente a la regla gramatical aplicada.

En el segundo nivel, se comprueban todos los *bytecodes* generados, y se adecúan los argumentos de los mismos. Una vez finalizado este proceso, el resultado es una lista de *bytecodes* listos para ser ejecutados.

Resulta un compilador sencillo ya que el lenguaje no permite expresiones complejas ni definición de variables ni retornar valores. Sí admite la definición de procedimientos con parámetros, aunque éstos son más bien estáticos y el programa los traduce a constantes

No obstante, internamente sí que utiliza variables, pero éstas son resultados, de la ejecución de procedimientos o, por ejemplo, los datos relativos a la posición del robot.

3.5. Máquina Virtual

Para la máquina virtual existe una interfaz, *VirtualMachine*, la cual define los métodos que han de implementarse. Es curioso que para la definición de esta interfaz, utiliza un elemento de la clase que la implementa, (que en este caso es *VisualVM)*, cuando una de las finalidades de las interfaces es que no es necesario conocer la clase particular que implementa dicha interfaz para utilizar cualquier instancia de la misma.

Por su parte, *VisualVM* tiene como atributos una pila de marcos, (clase anidada *Frame)*, y una pila de valores. Los marcos sirven para saber en cada procedimiento cual es el puntero del programa. El apilarlos tiene sentido para realizar llamadas a otros procedimientos, al llamarlos, se crea un marco y se apila, al retornar, se desapila y se continua con el marco siguiente. La pila de valores, es el resultado de la última consulta. Si se quiere realizar alguna acción, por ejemplo, si el frente es blanco, la máquina virtual traduce esto en: comprobar si el frente es blanco; si la ultima comprobación es verdadera saltar relativamente 1. Así, al ejecutar la

maquina virtual comprueba si el frente es blanco y apila, luego la comprobación desapila y salta si fue verdadera, o, si no lo fuere, hace un salto más largo hasta el final de la comprobación.

3.6. Esquema general

La figura 3.5.1 es un esquema UML resumido de las relaciones entre las clases arriba descritas, como esquema general del programa RoboMind. Se han omitido ciertas relaciones para simplificar y clarificar el esquema

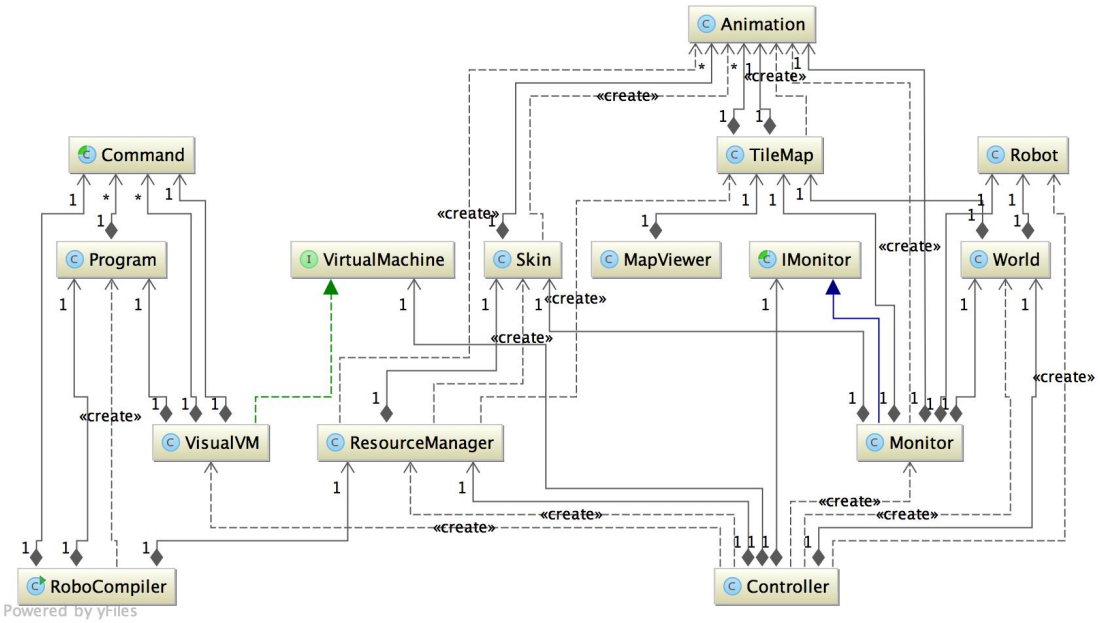

Figura 3.5.1. Esquema UML resumido del programa

4. EDITOR GRÁFICO DE MAPAS

En esta sección se pretende explicar con detalle el desarrollo del editor gráfico de mapas. En la primera parte, se explicará como se representan los mapas y sus elementos. En la segunda, los detalles de la implementación, así como los elementos del programa principal modificados para poder realizar la integración. Por último se detallan las pruebas aplicadas al programa para la comprobación de su correcto funcionamiento.

4.1. Ficheros de Mapas

Los ficheros de mapas de RoboMind son ficheros de texto ASCII con las extensión ".map". Pueden ser abiertos con cualquier editor de textos estándar.

Cada fichero contiene una descripción del mundo que representa en la que cada carácter simboliza una celda dentro del mapa. A estas piezas se les llaman *tiles*, azulejos en español.

Además de los *tiles* es posible añadir trazos de pintura. Mediante una lista de coordenadas, como se detallará más adelante.

Por último estos ficheros permiten añadir comentarios comenzando la línea con el carácter "#". La línea entonces será ignorada, y no afectarán al mundo representado.

4.1.1. Formato de mapa y posibles caracteres.

En la figura 4.1.1.1 se pueden observar todas las posibles *tiles* así como su carácter correspondiente. Esta es la apariencia que tienen las figuras cuando se utiliza la piel "desert" (piel por defecto). RoboMind permite cambiar el tipo de piel y diseñar pieles personalizadas:

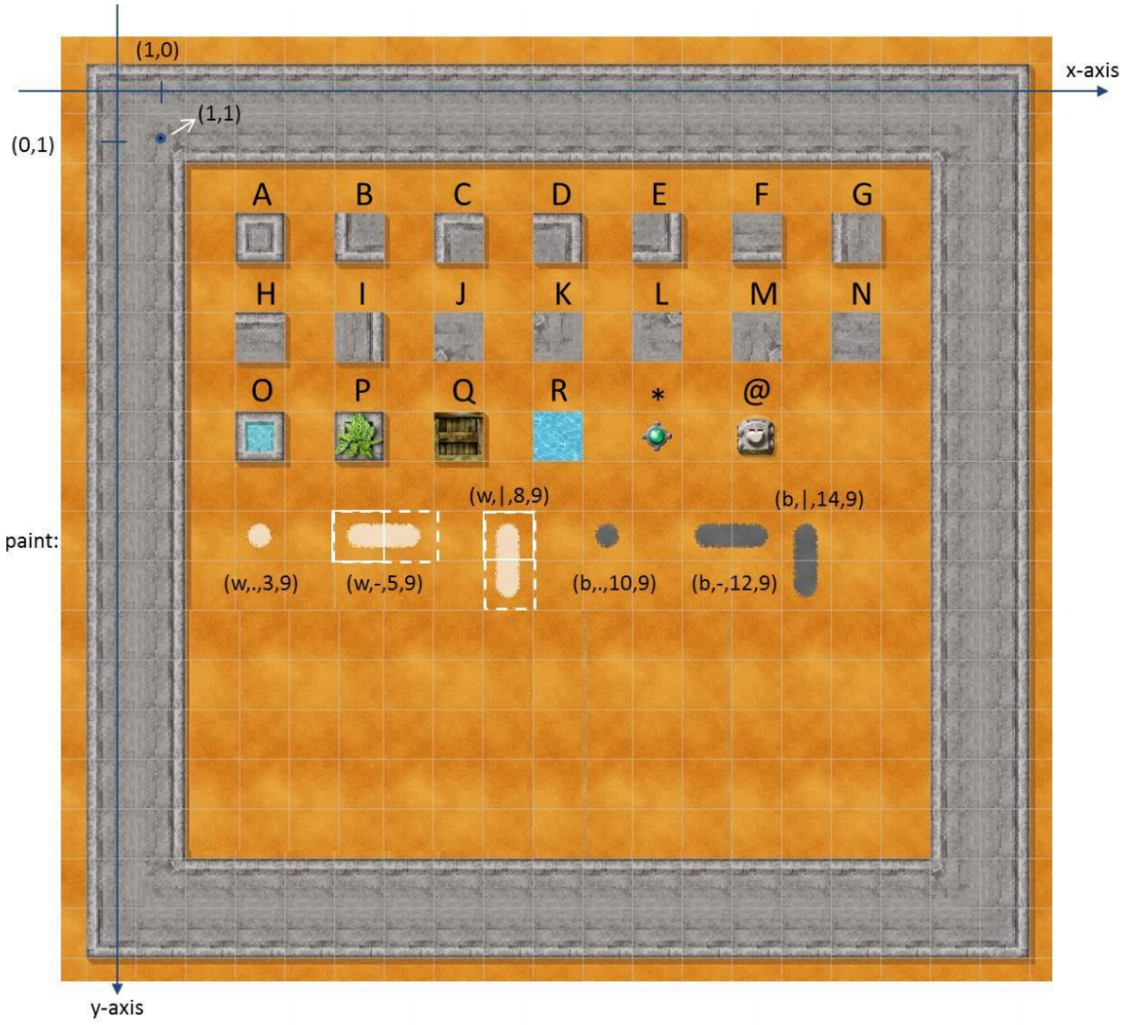

Figura 4.1.1.1. Mapa RoboMind

La figura anterior es el resultado de cargar un fichero de mapa en RoboMind. El contenido del fichero es el siguiente:

```
#map:def
# Example map file for RoboMind

paint:
(w,.,3,9)(w,-,5,9)(w,|,8,9)(b,.,10,9)(b,-,12,9)


map:
CHHHHHHHHHHHHHHHHD
GMFFFFFFFFFFFFFFJI
GI                GI
GI A B C D E F G GI
GI                GI
GI H I J K L M N GI
GI                GI
GI O P Q R * @    GI
GI                GI
GI                GI
GI                GI
GI                GI
GI                GI
GI                GI
GI                GI
GI                GI
GLHHHHHHHHHHHHHHHKI
BFFFFFFFFFFFFFFFFE
```

Como se puede observar, los ficheros están divididos en dos secciones delimitadas: la sección *paint* y la sección *map.* Los *tiles* en el mapa tienen coordenadas relativas a la parte superior izquierda que es el origen de coordenadas. El eje "X" crece de izquierdas a derechas y el eje "Y" de arriba abajo.

4.1.2. La sección map

Esta sección estructura los elementos del mapa, definidos después de la línea "map:". Éstos son:

- *Tiles,* representados por letras mayúsculas como se indica en el código 4.1.1.1.
- Espacios en blanco, correspondientes a una celda vacía
- Balizas, añadidas mediante el asterisco "*"
- La posición inicial del robot, añadiendo el símbolo "@"

4.1.3. La sección paint

Los trazos de pintura se pueden añadir al mapa luego de la línea "paint:", mediante una lista con el siguiente formato: (color, tipo, x, y) donde:

- Color: el color de la pintura, puede ser w para blanco o b para negro.
- Tipo: la forma del trazo, puede ser>
  - "." : un punto
  - "|" : para una línea horizontal hacia abajo (dos celdas)
  - "-" : para una línea vertical hacia la derecha (dos celdas)

Una vez descrito este formato, se explicará el desarrollo del programa.

4.2. Desarrollo

En esta sección se pretende analizar los pasos seguidos para el desarrollo del editor gráfico de mapas. Este editor se ha desarrollado en el paquete *robo.mapeditor*, en el que se ha incluido todas las clases necesaria para su funcionamiento. También ha sido necesaria la modificación de algunos de los ficheros fuente originales para la correcta integración del mismo.

En un primer momento, se pretendía realizar las fases por separado de desarrollo e integración. Finalmente se decidió no hacerlo de esta manera, al ser muy fuerte la dependencia con los elementos del programa original. Por lo tanto, se realizaron cambios en el código original para poder efectuar la integración y finalmente se procedió a su desarrollo. Una de las clases más necesitadas ha sido *Monitor*, ya que es la encargada de la visualización del mundo en pantalla.

A continuación se puede ver en la figura 4.2.1, el diagrama UML resumido de las clases desarrolladas y su relación con las clases del código. En las siguientes secciones, primero se explicarán y justificarán los cambios realizados en el código original, en segundo lugar se detallará el desarrollo llevado a cabo y, por último, las pruebas realizadas.

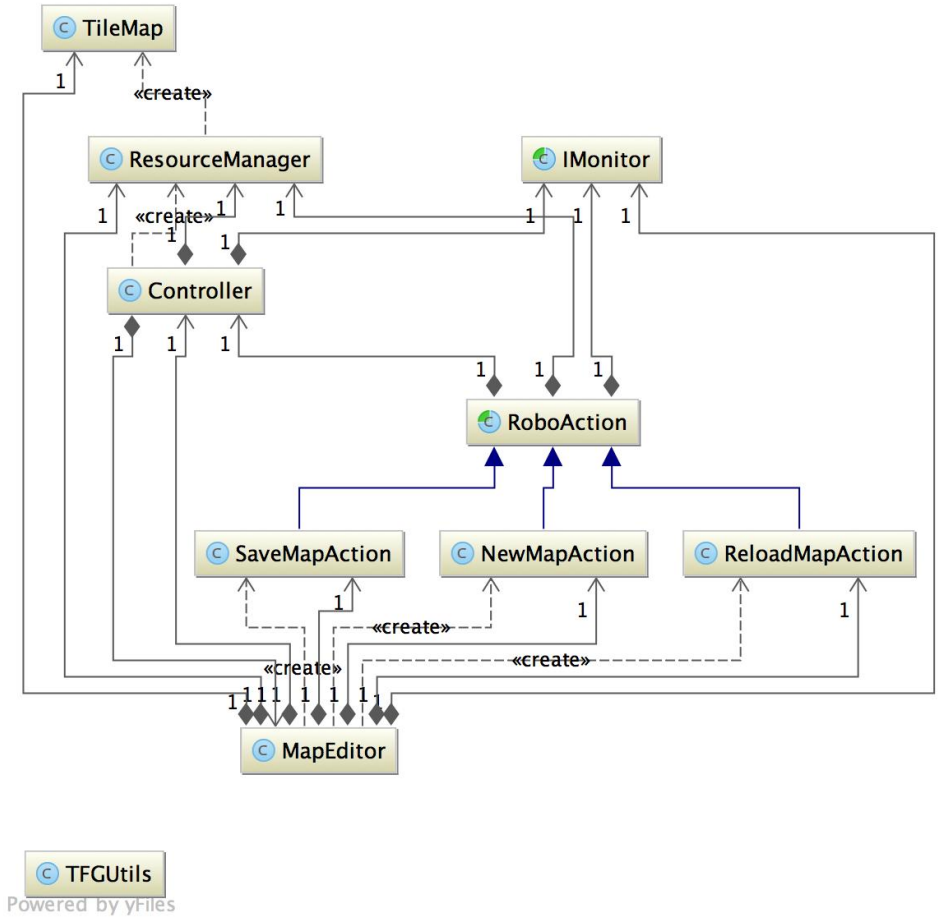

Figura 4.2.1. Diagrama UML del editor de mapas.

4.3. Funcionamiento del editor

El editor de mapas pretende ser una herramienta para que el usuario pueda editar y crear nuevos mapas directamente en la aplicación. El funcionamiento es simple: mediante una barra de herramientas, el usuario puede realizar diferentes acciones, como guardar el mapa o crear uno nuevo, y también seleccionar una de las herramientas para modificar directamente el mapa.

Las herramientas para modificar el mapa son las descritas en el apartado 4.1.1, en las que se describían las posibles *tiles* que podía contener.

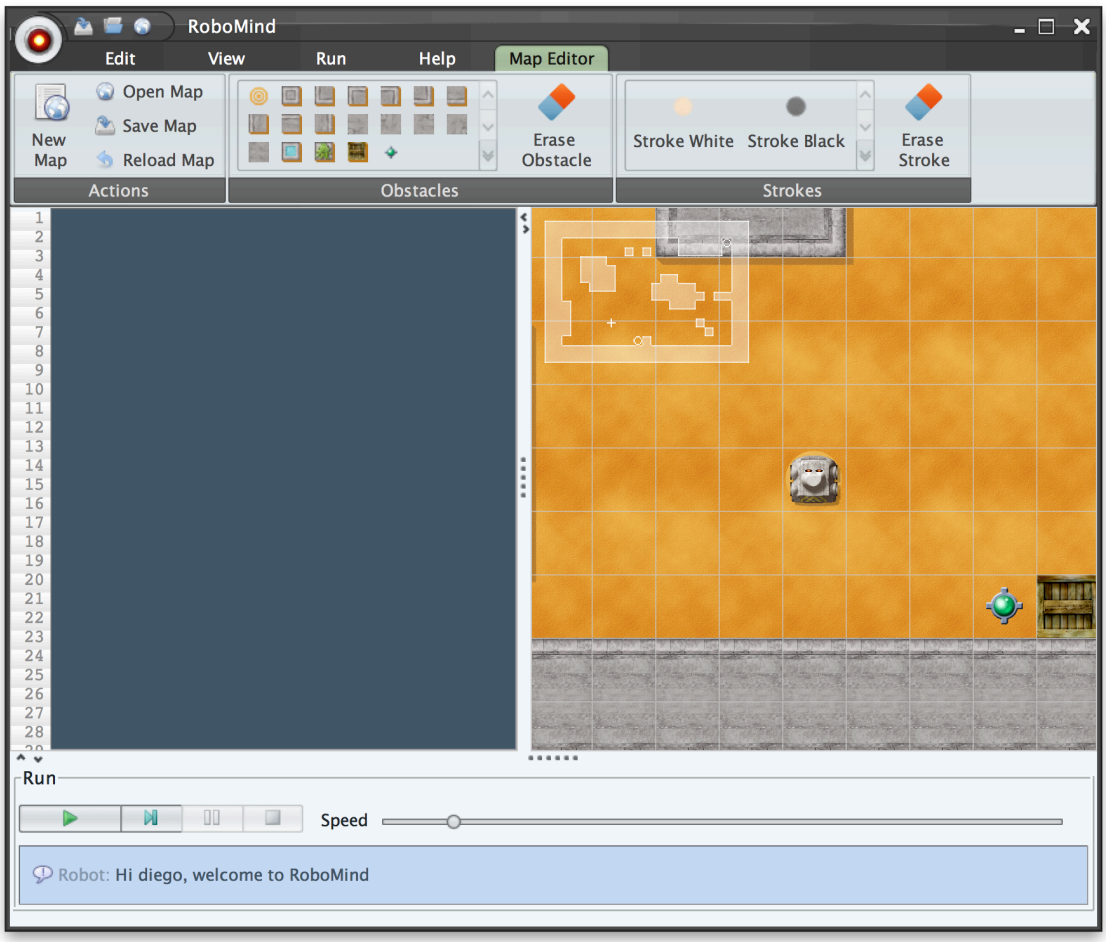

Figura 4.3.1. Programa con el editor de mapas integrado

Como se puede observar en la figura 4.3.1 se ha creado una barra de herramientas integrada en la apariencia del sistema. Esta barra de herramientas se ha construido mediante el uso de la librería *Flamingo*, que se estuvo desarrollando hasta el 2010, y cuya documentación es muy escasa. Mediante algunos ejemplos ha sido posible realizar el diseño. También notar que la representación del mapa que se ve en la figura, es la clase *Monitor,* que como se explicó en apartados anteriores, extiende a la clase *JPanel* lo que hace posible su visualización y también la interacción del usuario con éste.

La barra de herramientas está dividida en tres zonas: la zona de *Actions (*acciones), *Obstacles* (obstáculos) y *Strokes* (trazos). Las acciones son aquellas relacionadas con el fichero del mapa actual, permite crear uno nuevo, guardar y recargar el actual. Los obstáculos son elementos del mapa que tienen una relación directa con los vistos en el apartado 4.1.1. Del mismo modo, los trazos, son los mencionados en dicho apartado. En los dos zonas que modifican el mapa se encuentran dos borradores, que sirven para eliminar elementos del mismo grupo.

Una vez que se activa la pestaña correspondiente al editor de mapas, el editor se pone en modo edición (*editing*), que afecta al comportamiento de la clase *Monitor*, debido a las modificaciones que se explicarán en secciones posteriores. El usuario puede seleccionar cualquiera de las herramientas y utilizarlas directamente en el mapa. El editor tiene en consideración ciertos elementos no pueden estar encima de otros (por ejemplo, donde esté localizado el robot no puede ir ningún obstáculo). Además el editor permite presionar el botón del ratón en el mapa y arrastrar, por lo que la herramienta seleccionada se aplicará a lo largo del recorrido del ratón y hasta que se suelte el botón, y realizando el desplazamiento oportuno si se llega a cualquier límite del *Monitor*.

Todos los cambios realizados en el mapa van actualizando las estructuras internas del programa. Se ha desarrollado un método que permite leer estas estructuras y traducirlas al formato de salida de mapas de robomind.

4.4. Detalles de implementación

El desarrollo del editor se centra en la clase *MapEditor,* de la cual vamos a comentar los aspectos más importantes.

Al instanciarse el editor, lo primero que realiza es la obtención de las instancias de las clases que controlan el programa RoboMind, (*Controller, Monitor* y *ResourceManager).* Una vez finalizado esto, se crean las acciones para responder al la interacción del usuario, y esta clase se subscribe a los eventos del ratón de *Monitor.* De esta manera, cuando el usuario interactúe con *Monitor*, *MapEditor* podrá realizar las acciones oportunas en el caso de que esté en modo edición. También en esta fase se construye la barra de herramientas.

El programa tiene dos modalidades de función: mediante el uso de las barras de tareas llamadas *Ribbon,* las cuales han sido muy populares desde que Microsoft las sacara por primera vez en Microsoft Office 2007, y más tarde extendiéndola a todo el sistema operativo Windows; la otra modalidad de funcionamiento es empleando las barras de herramientas convencionales. El constructor de *MapEditor* tiene en cuenta en qué modo se ha iniciado el programa, y determina que tipo de barra de herramientas tiene que crear. En la figura 4.3.2 se puede observar el programa funcionando con la barra de herramientas tradicional.

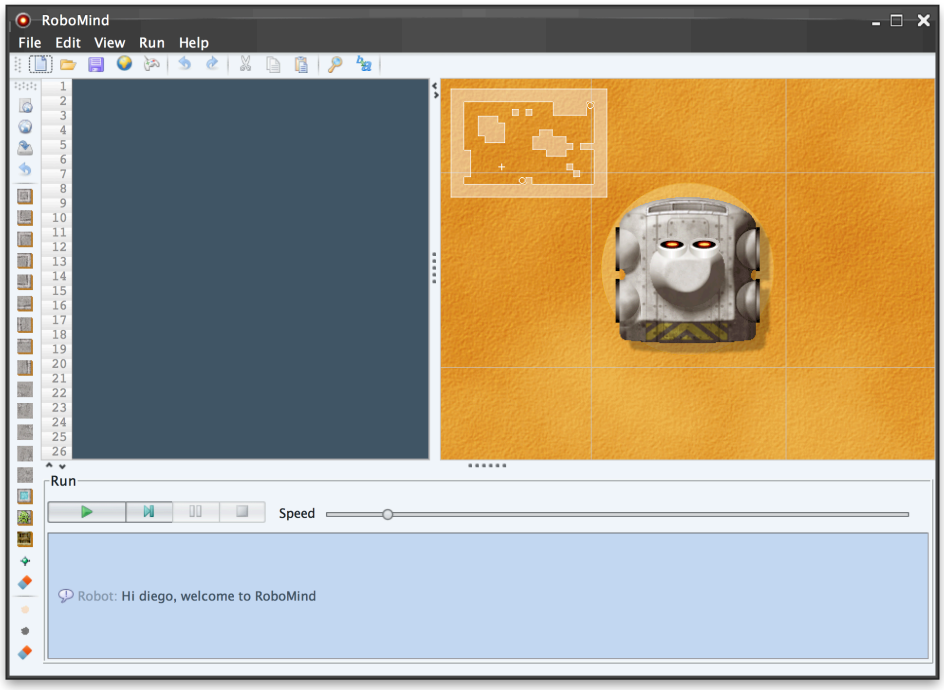

Figura 4.3.2. *Programa funcionando con la barra de herramientas tradicional*

Como se puede observar en la figura anterior, la barra de herramientas se sitúa en el lado izquierdo de la pantalla. Aunque puede ser posicionada en cualquiera de los dos lados. El usuario puede decidir que barra de herramientas utilizar accediendo a las preferencias del programa.

Cuando el editor recibe la acción del ratón, el despachador de eventos de Java invoca uno de los métodos relacionados con el ratón según el tipo de evento: un clic, un arrastre, ... Todos los eventos del ratón se producen con un objeto *MouseEvent* el cual describe las coordenadas donde se ha hecho el clic. Estas coordenadas son relativas a *Monitor* y no son directamente tuplas (x, y) del mapa. Por lo que se ha implementado el método *translate:*

```
489 private Point translate(double mouseX, double mouseY) {
490 
491         // Required variables.
492         double scale = monitor.getScale();
493         Point2D t = monitor.getTranslation();
494         Point2D g = monitor.getGridDimension();
495 
496         double tx = t.getX(), ty = t.getY();
497         double gridDx = g.getX(), gridDy = g.getY();
498 
499          // First scales x,y position.
500          double cx = mouseX / scale, cy = mouseY / scale;
501 
502          // Now, we translate it and divide it by grid size
503          double x = ((cx - tx) / gridDx), y = ((cy - ty) /
                                                        gridDy);
504 
505         // In case of negative values, we substract 1 so
               there is only
506         // one zero, ie: -0.005 -> truncated is 0, when
               actually we want -1
507         x = x < 0 ? x - 1 : x;
508         y = y < 0 ? y - 1 : y;
509 
510         // Values are truncated.
511         return new Point((int) x, (int) y);
512     }
```

Lo que este código realiza es lo siguiente: primero lee el contexto actual del mapa, la traslación, la escala y las dimensiones en pixeles de cada celda, valores que no están sometidos a escala; en segundo lugar, divide las componentes x e y del punto por la escala, obteniendo en qué punto (x, y) habría cliqueado el usuario a escala 1; luego se le resta la traslación actual a cada componente. Aquí es donde aparece el problema de los números negativos. Un número negativo implica que se ha hecho clic fuera de los límites del mapa. El problema es que para obtener coordenadas discretas (x, y) hay que truncar los valores obtenidos, y truncar, por ejemplo -0.23, es 0, cuando en realidad el usuario ha hecho clic en la celda -1, por lo tanto, antes de truncar el valor, hay que restarle una unidad, para solucionar este problema.

También se ha desarrollado un método que comprueba los límites del mapa. Aunque la capacidad de desplazamiento del mapa es muy grande, las estructuras internas del programa solo almacenan los datos de las partes del mapa que contienen datos. Así, si la celda con algún elemento más lejana es la (200, 100), las estructuras internas tendrían esas dimensiones. El método *Point checkBounds(Point point)* comprueba si el punto *point* está dentro de los límites del mapa. De lo contrario realiza todas las acciones necesarias para que ese punto sea parte del mapa, ampliando las estructuras internas.

Esta tarea, en principio puede parecer trivial, pero no lo es. Si el punto se es negativo, parte de ampliar las estructuras, habría que desplazar todos los elementos de las mismas para que sigan en el sitio que les corresponde; es decir, actualizar su posición. Además, el monitor guarda muchos de los elementos del mapa en estructuras propias, que no son las del mapa actual, por lo que ha sido necesario incluir un método en el monitor que mueva los elementos visuales de sus estructuras. Otro problema que surge en este caso es que la manera que pinta el fondo (que es como arena), es relativo a la traslación (que se ve modificada al

ampliar el mapa), por lo que hay que aplicar un factor de corrección para que el fondo se dibuje de manera correcta.

El editor reacciona a los siguientes eventos del ratón:

- *mousePressed(MouseEvent e)*: este método se invoca al presionar un botón del ratón. El editor aprovecha en este evento para tomar las coordenadas donde se ha hecho clic. Esto es importante ya que al arrastrar el ratón hay que saber desde donde hay que aplicar la herramienta seleccionada. Este valor se guarda como coordenadas del mapa y no como coordenadas en el *Monitor*.

- *mouseClicked(MouseEvent e):* este método se invoca cuando el usuario realiza un clic. Este método es el responsable de aplicar la herramienta seleccionada sobre el mapa. Según de qué herramienta se trate actualizará las estructuras para añadir dicho objeto en el mapa. Además, comprobará los límites del mapa, y comprobará las restricciones de alguno de ellos. Algunas de estas restricciones son: no se puede borrar el punto de inicio del robot; no se puede insertar un objeto sobre el punto de inicio del robot. También actualiza el radar, que es una imagen traslucida con una miniatura del mapa.

- *mouseDragged(MouseEvent e):* este método se invoca cuando el usuario arrastra el ratón con el botón apretado. Si la herramienta seleccionada es un obstáculo, entonces llama a *mouseClicked. E*n caso de que la herramienta sea un trazo, entonces dibuja el trazo desde la última posición en la que se produjo el evento. Una vez pintado, se actualiza el último punto donde se realizó el arrastre para poder actualizar el próximo evento.

- *mouseReleased(MouseEvent e)*: este método ocurre cuando se deja de presionar el botón del ratón. En este método sólo se deja la referencia al último punto

Para que todo esto funcione y pueda ser integrable en el sistema, ha sido necesaria la modificación de algunos ficheros del código original, los cuales se detallan en la siguiente sección.

4.5. Cambios en el código original

Las modificaciones en el código original han sido puntuales y por causas justificadas. En cada cambio realizado se ha comentado la línea o bloque en la que se ha producido, junto con su justificación. Un ejemplo sería el siguiente cambio realizado en la clase *Controller*:

```
58 private MapEditor mapEditor; // [TFG] MapEditor
                                   Instance.
```

Como se puede observar, se utiliza la etiqueta *[TFG]* junto con la descripción. De esta manera, utilizando una herramienta como *grep* se podrían encontrar todos los cambios efectuados. También, dado que no se ha realizado ninguna otra modificación más que los comentarios, se pueden utilizar herramientas como *diff* para comprobar los ficheros original y nuevos y ver los cambios.

Las clases modificadas han sido las siguientes: *Controller, IMonitor, Monitor y NewMonitor, ResourceManager* y *TileMap.*

4.5.1. Controller

La clase *Controller* ha tenido que ser modificada por los siguientes motivos:

- Es necesario que *Controller* instancie el editor de mapas cuando realiza la construcción de la interfaz gráfica para que el editor esté activo.
- Es necesario cambiar la visibilidad de algunos atributos para poder acceder a ellos fuera de la clase *Controller*
- Es necesario que *Controller* compruebe si el mapa actual ha sido modificado antes de realizar cualquier acción relacionada con el uso del mapa (como ejecutar un *script* o salir del programa) y preguntarle al usuario qué desea hacer (guardar cambios, descartarlos, cancelar).
- Es necesario que *Controller* añada la nueva barra de herramientas para poder manipular el editor de programas.

Por lo tanto los cambios realizados son los siguientes:

```
48public class Controller extends WindowAdapter implements
                                                ComponentListener {
49    private static Controller thisController;
50    private ResourceManager resourceManager;
51//    private PluginManager pluginMananger;
52    private MapEditor mapEditor; // [TFG] MapEditor Instance.
53    private JToolBar toolBar; // [TFG] ToolBar Instance
```

En este trozo de código podemos observar los nuevos atributos añadidos. El atributo *mapEditor* es la instancia actual del editor de mapas, que *Controller* será el encargado de crear; el atributo *toolBar* es la barra de herramientas del editor de mapas.

```
141private Controller(String[] args, boolean useSubstance) {
142        thisController = this;

              ...

231        /* [TFG] Instanciates MapEditor */
232        mapEditor = MapEditor.getInstance();
233
```

```
234     }
```

En este caso, al finalizar la construcción de *Controller* se añaden las líneas para instanciar el editor de mapas.

```
246     /*
247      * [TFG] Changed Scope from private -> public
248      * we take advantage of this method to show errors
                                                    anywhere.
249      */
250     public void showError(Object msg){
```

Se ha cambiado la visibilidad de este método para que pueda ser utilizado fuera del ámbito de la clase. De esta manera en el caso de tener que mostrar algún error al usuario, este mensaje va a tener el mismo formato que el resto del programa.

```
418     /**
419      * Create a gui with the just initialized actions
420      */
421     private void initGui(){
              ...
479        else{
480            // Create Menu
481            frame.setJMenuBar(createMenuBar());
482            // Create Toolbar
483            toolBar = createToolBar(); // [TFG] toolbar
484            container.add(toolBar, BorderLayout.NORTH);
485        }
              ...
1418
1419    /* [TFG] Added getter for toolbar */
1420    public JToolBar getToolBar() {
1421        return toolBar;
1422    }
```

En este caso se instancia la barra de herramientas al atributo añadido anteriormente. También se han creado el método *getter* del mismo.

```
1707        public void openMap(File file) {
1708            try {
1709                /* [TFG]  before open a new map, we maybe want
                                                        to save changes
1710                  * made on current */
1711                if (mapEditor.maybeSaveCurrentFileMap()){
1712                    return;
1713                }
1714                /*    *    */
1715                TileMap map = resourceManager.loadMap(file);
1716                /* [TFG]  */
1717                resourceManager.setMapFile(file); /* [TFG]
                                            sets current map file */
1718                mapEditor.setMap(map);
1719                /*    *    */
                    ...
```

En este código, se comprueba antes de abrir otro mapa, si el actual tiene cambios, y se le pregunta al usuario qué desea hacer. Mediante la función *maybeSaveCurrentFileMap()*. En el caso de que el usuario desee cancelar, se devuelve *true.* Una vez abierto el nuevo mapa, se le envía el mensaje a *mapEditor* con el nuevo mapa.

El resto de modificaciones son las que se encargan de comprobar si hay que preguntar al usuario si se debe guardar el mapa. Éstas están especificadas en el Anexo A.

4.5.3. IMonitor, Monitor y NewMonitor

Monitor es una de las clases más utilizadas por el editor de mapas, ya que es la encargada de responder ante los eventos del usuario hacia el mapa. Muchos de los modificaciones realizados son del tipo de creación de *getter* y *setters* así como de cambios de visibilidad de métodos, necesarios para el editor. Otros cambios sí son

más específicos y transforman algo el funcionamiento de *Monitor,* aunque de una manera transparente al resto del programa. Es decir, si el editor de mapas no existiese, a pesar de estos cambios, el resto del programa seguiría funcionando de la misma manera que lo hacía antes. Las modificacioens se pueden observar en el anexo B.

Comentar que *NewMonitor* es una clase en desuso, una de las primeras clases desarrolladas de la implementación de *IMonitor*, pero que ha sido reemplazada por *Monitor*, dado que *NewMonitor* intentaba mejorar el rendimiento gráfico evitando escalar las imágenes cuando no era necesario. Dado que a partir de Java 6 ya se tienen aceleración de gráficos de forma nativa, esta clase ya no tiene sentido. Además, parece ser que *NewMonitor* tiene algunos *memory-leaks*, ya que las imágenes escaladas no son liberadas después de su uso. Dado que los atributos en *NewMonitor* son los mismos que en *Monitor* y su funcionamiento también es coincidente, se han aplicado los mismos cambios tanto en *Monitor* como en *NewMonitor.*

El cambio más importante se realiza en el método *mouseDragged* de la clase *Monitor,* para otorgarle más usabilidad al programa (dada la extensión de esta modificación, el código se puede obtener en el anexo C). En el modo edición, cuando se arrastra el ratón y se llega a los límites de *Monitor*, hay que modificar la traslación del mapa para seguir utilizando la herramienta. Esto se ha hecho de la siguiente manera:

- Se ha definido un atributo ***private long*** *last*, donde se guarda el milisegundo de la última vez que se realizó el arrastre. Al producirse el evento *mouseDragged*, se vuelve a tomar el tiempo. Sólo se realizará el arrastre si la diferencia entre dos eventos está dentro de un límite. Estos límites se han calculado según la distancia entre el ratón y el límite del *Monitor.* La política en este caso es que mientras más cerca esté del

límite más rápido será el desplazamiento, por lo que la variación de tiempo entre eventos es menor. Si no se controlaran los límites, desplazándose cada vez que se produjese el evento y se estuviera cerca del límite de *Monitor*, entonces este desplazamiento sería mucho más de lo deseado haciendo reduciendo la usabilidad, al producirse los eventos de manera muy continua.

- En caso de que las variaciones de tiempo se encuentren dentro de los límites, se realizan los cálculos para el desplazamiento del mapa y se ordena que repinte todo el *Monitor*. También se actualiza el valor de la variable *last*. En caso negativo, no se modifica nada.

4.5.4. ResourceManager

El primer problema que hubo a la hora de realizar la implementación del editor, fue que todo el programa busca las cadenas de texto visibles al usuario en un *ResourceBundle* que varía según el idioma del programa. Esto se hace mediante el método *getLabel* de *ResourceManager,* al cual se le pasa como parámetro una clave que buscará dentro del *ResourceBundle*. En el caso de no existir dicha clave, arroja una excepción. Por lo que la primera modificación fue que en vez de arrojar una excepción, éste devolviera la clave pasa como parámetro.

```
283     public String getLabel(String key){
284        try{
285            return labelsBundle.getString(key);
286        }
287        catch (Exception e){
288            //return defaultLabelsBundle.getString(key);
289            /* [TFG] Refactored */
290            try {
291                return defaultLabelsBundle.getString(key);
292            } catch (Exception ex) {
293                return key;
294            }
295            /* [TFG] End */
```

```
296             }
297         }
```

También hubo que modificar el método *saveSnapShot*, el cual toma una instantánea del estado actual del mundo y la guarda en el formato JPEG. Para realizarla, hacía uso de unas librerías Java que ya no estaban disponibles, por lo que ese código ha sido reescrito para poder compilar todo el programa sin errores.

Por último, se ha añadido el método para guardar el mapa actual en un fichero. Este método, mediante las estructuras internas del mapa pasado como parámetro, organiza el fichero salida para que sea reconocible por el programa. Hay que recalcar, que *TileMap* tiene las estructuras preparadas para que este método no tenga más que iterar sobre ciertas estructuras, en concreto dos: *strokes* y *imageKeys,* y guardarlo en el fichero.

4.5.5. TileMap

Las modificaciones en esta clase han sido variadas, las más importantes pueden consultarse en el anexo D. Se han añadido *getters* y *setters* para atributos necesarios como: char[][] *imageKeys,* que contiene el carácter equivalente en cada tupla (x, y); *boolean modified*, el cual indica si se han producido modificaciones en el mapa.

El resto de modificaciones tiene que ver con las estructuras internas de la clase, la cual, tiene un método, *restore,* que restaura el mapa a su estado anterior. Este método fallaba en caso de haber modificado las estructuras internas, sobre todo por la ampliación de las dimensiones del mapa. Así que ha tenido que ser reescrita por completo.

La otra modificación importante ha sido en el método *updateStroke,* que ha sido reescrita para poder añadir nuevos trazos, teniendo en cuenta los ya existentes, para que no haya repetidos y que al guardar el mapa en un fichero, la lista de tuplas de las sección *paint* sea concordante. Para esto también ha sido necesario implementar los métodos *equals* y *hashCode*, en el la clase anidada *Stroke*, para que, de este modo, se pudiera comprobar si un *Stroke* ya existe o no. Al añadir una traza nueva, el método comprueba no solo si existe esa traza, sino también si existe alguna otra que pudiera pintar la misma ya que las trazas horizontales y verticales ocupan dos celdas.

Por último, se ha implementado el método *toString* de *Stroke*, de modo que devuelva una cadena con el formato que se requiere en los ficheros, facilitando la tarea de guardar el fichero de mapa.

4.6. Pruebas

Las pruebas que se han realizado han sido manuales, al requerir que se dibuje un mapa para ver que el resultado sea el esperado. En primer lugar, se ha probado con todos los ficheros de mapas que proporciona la aplicación como ejemplo, se han abierto, se han modificado, se han guardado, se han cerrado y por último se han vuelta a abrir para comprobar que efectivamente, los mapas eran iguales.

También se crearon mapas desde cero, y se comprobó el resultado del fichero de salida, y, por supuesto, se volvieron a abrir para verificar que fuera el mapa creado.

En las figuras siguientes, se puede ver un mapa creado manualmente, copiando uno de los ejemplos de la página oficial de RoboMind, con su correspondiente salida del fichero. Nótese que el comentario inicial es estándar para todos los

mapas guardados con el editor de mapas, por lo que cualquier comentario que tuviera el fichero abierto, al guardarlo se reemplazaría por el genérico:

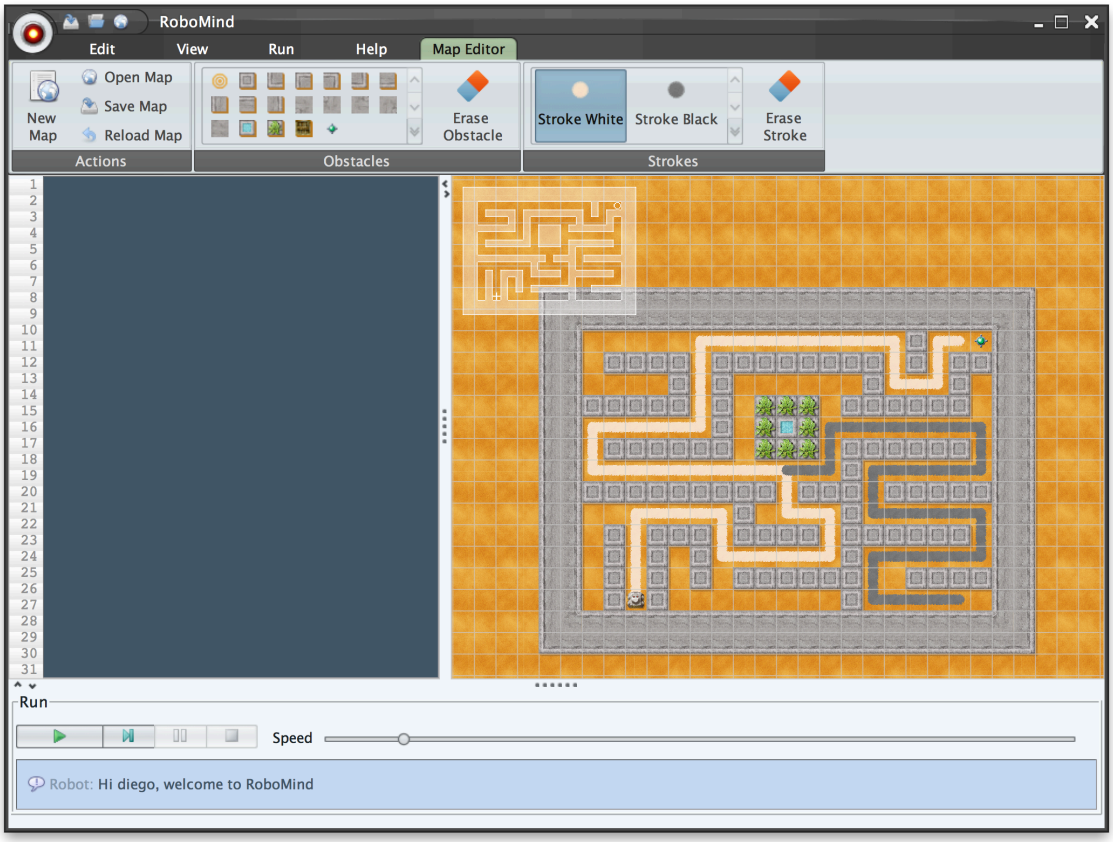

Figura 4.6.1. *Mapa creado con la herramienta.*

```
# This file has been automatically generated
paint:
{(w,|,4,12)(w,|,4,13)(w,|,4,11)(w,|,4,10)(w,-,4,10)(w,-
,7,10)(w,|,8,10)(w,-,6,10)(w,-,5,10)(w,|,8,11)(w,-
,8,12)(w,-,9,12)(w,-,10,12)(w,-,11,12)(w,-
,12,12)(w,|,13,11)(w,|,13,10)(w,-,12,10)(w,-
,11,10)(w,|,11,9)(w,|,11,8)(w,-,9,8)(w,-,10,8)(w,-,8,8)(w,-
,7,8)(w,-,5,8)(w,-,6,8)(w,-,2,8)(w,-,3,8)(w,-
,4,8)(w,|,2,7)(w,|,2,6)(w,-,2,6)(w,-,3,6)(w,|,7,5)(w,-
,6,6)(w,-,5,6)(w,-,4,6)(w,|,7,3)(w,|,7,2)(w,|,7,4)(w,-
,7,2)(w,-,8,2)(w,-,9,2)(w,-,15,2)(w,-,12,2)(w,-,13,2)(w,-
```

```
,14,2)(w,-,11,2)(w,-,10,2)(w,|,16,2)(w,|,16,3)(w,-
,16,4)(w,-,17,4)(w,|,18,3)(w,|,18,2)(w,-,18,2)(b,-
,12,8)(b,|,13,7)(b,|,13,6)(b,-,13,6)(b,-,14,6)(b,-
,15,6)(b,-,16,6)(b,-,17,6)(b,-,18,6)(b,-,19,6)(b,-
,18,8)(b,-,16,8)(b,-,17,8)(b,-
,15,8)(b,|,20,6)(b,|,20,7)(b,-
,19,8)(b,|,15,8)(b,|,15,9)(b,-,15,10)(b,-,16,10)(b,-
,17,10)(b,-,18,10)(b,-,19,10)(b,|,20,10)(b,|,20,11)(b,-
,19,12)(b,-,18,12)(b,-,17,12)(b,-,15,12)(b,-
,16,12)(b,|,15,12)(b,-,17,14)(b,-,15,14)(b,-,16,14)(b,-
,18,14)(b,|,15,13)(b,-,11,8)}
  map:
  CHHHHHHHHHHHHHHHHHHHHD
  GMFFFFFFFFFFFFFFFFFFJI
  GI                A  *GI
  GI AAAA AAAAAAAA A AAGI
  GI    A A       A   A GI
  GIAAAAA A PPP AAAAAA GI
  GI      A POP        GI
  GI AAAAAA PPP AAAAAA GI
  GI            A      GI
  GIAAAAAAAAA AAA AAAAAGI
  GI       A    A      GI
  GI A AAA AAAA AAAAAA GI
  GI A A A      A      GI
  GI A A A AAAAAA  AAAAGI
  GI A@A        A      GI
  GLHHHHHHHHHHHHHHHHHHKI
  BFFFFFFFFFFFFFFFFFFFFE
```

Al volver a abrir el fichero se puede comprobar que, efectivamente, los mapas son iguales. También se han realizado pruebas con cada una de las herramientas,

verificando su funcionamiento, y se ha comprobado que el programa respondiera correctamente a los eventos del usuario.

Otras pruebas fueron las de comprobar que el modo edición se desactivara cuando no se encontrara activada ninguna herramienta o, en el caso de *JRibbon*, cuando no estuviera seleccionada la sección de Editor de Mapas.

Todas estas pruebas han ayudado a depurar el programa para llegar a un funcionamiento correcto.

5. EDITOR VISUAL DE PROGRAMAS

Uno de los objetivos de este proyecto era la creación de un editor visual de programas. Para dicho fin, se requería la manipulación de las estructuras internas del programa mediante un entorno gráfico de tal manera que el usuario pudiera manipular elementos que modificaran dicha estructura. Un ejemplo de esta programación se puede observar en la figura 5.1. Se trata de una aplicación web de Google denominada Google Blocky, este permite una programación visual para resolver laberintos.

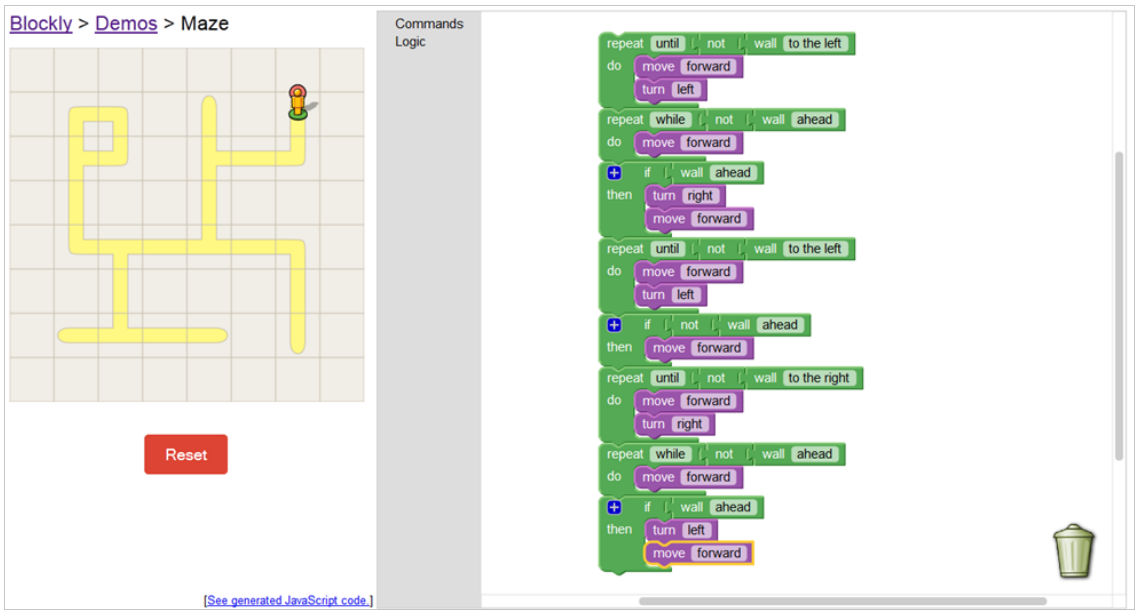

Figura 5.1. *Google Blocky en acción*

El problema con el que nos encontramos es que, como se comentaba en secciones anteriores, la estructura que almacena el programa a ejecutar es una lista de *bytecodes*, manipular este tipo de información mediante un editor visual sería no solo inadecuado, sino inviable.

En las secciones ulteriores, describiremos una posible solución a este problema, definiendo las estructuras que serían necesarias, así como un posible

diseño del mismo, basado en un compilador *Just-In-Time* con una máquina virtual basada en estructuras arborescentes. No se ha implementado dicha solución, ya que el tiempo necesario para tal objetivo excedía del tiempo disponible para todo el presente trabajo.

5.1. Planteamiento de la solución

Como se comentaba anteriormente, la estructura de datos por excelencia y óptima para este tipo de problemas, son los árboles. En concreto estamos hablando de los árboles sintácticos abstractos, en inglés AST. Estos árboles, tienen como raíz un único nodo, el cual tiene uno o más nodos hijo. Cada nodo hijo es un elemento del lenguaje, puede ser una asignación, un bucle, una condición, etc.

Figura 5.1.1. Árbol Sintáctico Abstracto.

Este tipo de estructura tiene una característica, que es que son fáciles de ejecutar. Su ejecución estaría en recorrer en profundidad este árbol, interpretando cada tipo de nodo.

Consideremos el siguiente ejemplo escrito en RoboMind:

```
1 if (frontIsClear() and flipCoin()) {
2         forward(1)
3 }
```

En este caso de condicional *IF*, uno de los hijos sería la condición a comprobar, la otra qué hacer en caso de cumplirse la condición, este nodo sería en sí un subárbol AST, ya que las acciones que se pueden realizar son ninguna, una o varias. En la figura 5.1.2 podemos ver la representación gráfica de este ejemplo.

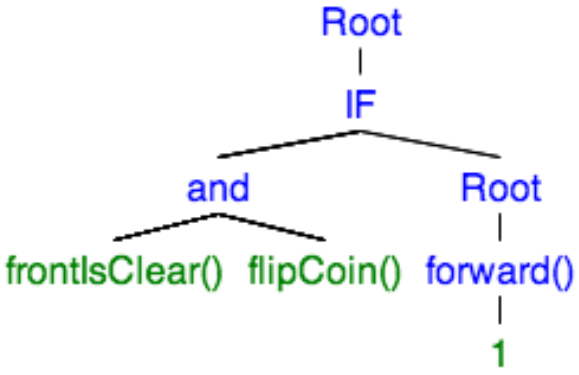

Figura 5.1.2. *AST ejemplo*

Se ha creado en el paquete *robo.programeditor,tree* algunas clases sobre posibles implementaciones de algunos de los nodos. Dado que se trata de un lenguaje reducido, no habría demasiados tipos de nodos. Además habría una relación directa entre el lenguaje de programación y la estructura interna que alberga dicha los programas, por lo que sería viable la construcción de un entorno visual que manipulara dicha estructura. Asimismo, utilizando clases serializables

(interfaz *Serializable*), se podría guardar una representación en un fichero que también se correspondiera con dicho árbol y, utilizando las librerías que proporciona Java, podría guardarse en un formato estándar, como es XML.

Otra de las características que puede tener esta parte, y que sería muy necesaria, es la de incorporar la posibilidad de exportar o importar scripts de RoboMind a este editor visual.

La exportación sería relativamente trivial, dado que el árbol tiene una relación directa con el lenguaje. En el caso inverso, sería un poco más complicado. Habría que definir una gramática del lenguaje, y construir un compilador. Este compilador tendría sólo los módulos del *lexer* o analizador léxico, y *parser* o analizador sintáctico. El analizador sintáctico tendría que, además de comprobar la sintaxis, ir creando el AST dimanante del código analizado.

El último paso en esta fase, sería la construcción de la máquina virtual, o más bien módulo de ejecución, ya que el nombre de máquina virtual queda reservado a la utilización de *bytecode*, gestión de memoria, etc. En cualquier caso, se debería implementar la interfaz *VirtualMachine*, para poder integrar el sistema con el resto del programa.

Este último paso, sería el encargado de manipular *Monitor*, para mover el robot, algo que es muy sencillo dado que *Monitor* contiene todas los métodos necesarios para lograr tal objetivo.

6. CONCLUSIONES Y LÍNEAS FUTURAS

Una vez acabado el trabajo, se dispone de una aplicación cuya funcionalidad de edición de mapas está perfectamente integrada en el mismo, además de toda una documentación de la herramienta, y la posibilidad de seguir ampliando el mismo.

Una de las partes más importantes es la vista atrás, comprobar todo lo aprendido, las dificultades que se han tenido, y la valoración del mismo.

De los objetivos planteados en un principio, éstos no se han podido cumplir todos, sin embargo se han planteado las soluciones para aquellos objetivos faltados. Futuros proyectos podrían seguir por la línea aquí planteada para obtener un programa realmente interesante.

6.1. Conclusiones personales

Tengo que decir que el proyecto era bastante ambicioso, y sobre todo que he podido aplicar muchos conocimientos adquiridos a lo largo de los años de estudio en un proyecto concreto. Sin embargo, me hubiera guastado haber aplicado más conocimientos en el desarrollo del objetivo que no se ha cumplido, si bien es cierto que la estructura en la que está realizada el programa no lo permite.

También tengo concluir lo importante de realizar una programación ordenada y metodológica. Tratar con código ajeno ha sido toda una experiencia, sobre todo en este caso, que era un código muy poco comentado y, a veces, un poco ofuscado. Es aquí cuando uno se da cuenta de la importancia de esto, sobre todo cuando sabes que tu trabajo puede ser utilizado por otros. Queda claro, que en caso de distribuir el código, se agradece una documentación que sea breve pero directa, diagramas UML, comentarios, etc.

En cuanto a las dificultades encontradas, han sido sobre todo las procedentes del análisis del código. Dada mi experiencia en aplicaciones *Java* y *Swing*, desarrollar la aplicación en sí no ha supuesto más problemas.

6.2. Líneas futuras

Al finalizar el proyecto quedan varias líneas de desarrollo, entre ellas:

- Mejora de la fluidez del editor de mapas

- Desarrollo de nuevas herramientas para editar mapas, como por ejemplo la posibilidad de crear rectángulos, teniendo en cuenta el tipo de pared que se tiene que utilizar en cada punto del mapa

- Implementar un algoritmo de generación automática de mapas. RoboMind está hecho para que se resuelvan problemas utilizando la programación, por lo que se podría generar mapas de distintos estilos, como laberintos, recogida de obstáculos, seguimiento de líneas, que pueden ser utilizados como base para que el usuario los modifique a necesidad.

- Por supuesto, el desarrollo del editor visual de programas.

A. OTRAS MODIFICACIONES EN CONTROLLER

```
1870public class ExitAction extends RoboAction{
1871        public ExitAction(String labelKey, String iconKey,
Integer mnemonic, KeyStroke accelerator) {
1872            super(labelKey, iconKey, mnemonic, accelerator);
1873        }
1874
1875        public void actionPerformed(ActionEvent e) {
1876            boolean cancel = maybeSaveCurrentFile() ||
mapEditor.maybeSaveCurrentFileMap(); // [TFG] Added map modify
check
1877            if(!cancel){
1878                task.cancel();
1879                System.exit(0);
1880            }
1881        }
1882    }
```

```
1969    public class ExecuteAction extends RoboAction{
1970        public ExecuteAction(String labelKey, String
iconKey, Integer mnemonic, KeyStroke accelerator) {
1971            super(labelKey, iconKey, mnemonic, accelerator);
1972        }
1973
1974        public void actionPerformed(ActionEvent e) {
1975            // [TFG]Condition added
1976            if (mapEditor.maybeSaveCurrentFileMap()) {
1977                return;
1978            }
              ...
2014                        monitor.setMap(map);
2015                        /*[TFG]*/ mapEditor.setMap(map);
```

```
2125 public class RemoteControlAction extends RoboAction{
2126        public RemoteControlAction(String labelKey, String
             iconKey, Integer mnemonic, KeyStroke accelerator) {
2127            super(labelKey, iconKey, mnemonic, accelerator);
2128        }
2129
2130        public void actionPerformed(ActionEvent e) {
2131            // [TFG] Condition added
2132            if (mapEditor.maybeSaveCurrentFileMap()) {
2133                return;
2134            }
```

B. CAMBIOS EN IMONITOR

```
99      /*[TFG] Required for MapEditor*/
100
101     public abstract double getScale();
102     public abstract java.awt.geom.Point2D getTranslation();
103     public abstract void updateRadar();
104     public abstract void setTranslation(double x, double y);
105     public abstract void setRelativeTranslation(double x,
                                                    double y);
106     public abstract java.awt.geom.Point2D
                                                getGridDimension();
107     public abstract void initBeakenSprites();
108     public abstract void setRoboSpritePosition(double x,
                                                   double y);
109     public abstract void moveSpritesRelative(double x,
                                                 double y);
110     public abstract void setCorrectorFactor(double x,
                                                double y);
111     public abstract java.awt.geom.Point2D
                                             getRoboSpritePosition();
112     public abstract robo.gui.Sprite getRobotSprite();
```

C. MODIFICACIÓN DE MOUSEDRAGGED EN MONITOR

```
935     public void mouseDragged(MouseEvent e) {
936         /* [TFG] Allows dragging  */
937 
938         if (MapEditor.getInstance().isEditingMode()) {
939             long ac = System.currentTimeMillis();   // Takes
                                                        current time
940 
941             int limit = 500;    // How long has to pass
                                                       before next dragg
942             // gridDx and gridDy are not scaled when zoom
                                                         in/out, so it's
943             // needed to scale them in order to know how big
                                                         grid cells are.
944             int dxs = (int) (gridDx*scale);
945             int dys = (int) (gridDy*scale);
946             // Performance stuff, no need to double check
947             boolean dxy = false;
948 
949             if (e.getX() <= (dxs/6) || e.getX() > (getWidth()
                                                     - (dxs/6))){
950                 // First speed (the fastest) 0 <= x <= dxs/6
951                 limit = 125;
952                 dxy = true;
953             } else if (e.getX() <= (dxs/3) || e.getX() >
                                          (getWidth() - (dxs/3))){
954                 // Second speed dxs/6 < x <= dxs/3
955                 limit = 250;
956                 dxy = true;
957             } else if (e.getX() <= (dxs/2) || e.getX() >
                                          (getWidth() - (dxs/2))){
958                 // Third speed (the slowest) dxs/3 < x <=
                                                           dxs/2
959                 limit = 500;
960                 dxy = true;
961             }
962 
963             // Same as before...
964             if (e.getY() < (dys/6) || e.getY() > (getHeight()
                                                     - (dxs/6))){
965                 limit = 125;
966                 dxy = true;
967             } else if (e.getY() < (dys/3) || e.getY() >
                                          (getHeight() - (dys/3))){
968                 limit = 250;
969                 dxy = true;
970             } else if (e.getY() < (dys/2) || e.getY() >
                                          (getHeight() - (dys/2))){
971                 limit = 500;
```

```
972                 dxy = true;
973             }
974
975             // in case it's not time of dragging or mouse is
                                        not on dragging limits
976             if ( (last != 0 && (ac - last) < limit)
977                     || !dxy){
978                 return;
979             }
980             // updates time, this is the last time user
                                      dragged successfully
981             last = ac;
982             // We only repaint once.
983             boolean rp = false;
984
985             if ( e.getX() <= dxs/2) {
986                 // Mouse is inside limits, move translation
                                            one cell right
987                 // and adjust it to grid boundary.
988                 tx = gridDx * (int)((tx + gridDx)/gridDx);
989                 rp = true;
990             }
991
992             if ( e.getY() <= dys/2 ) {
993                 // Mouse is inside limits, move translation
                                             one cell down
994                 // and adjust it to grid boundary.
995                 ty = gridDy * (int)((ty + gridDy)/gridDy);
996                 rp = true;
997             }
998
999             if (e.getX() >= getWidth() - (dxs/2)){
1000                // Mouse is inside limits, move translation
                                            one cell left
1001                // and adjust it to grid boundary.
1002                // In this case translation is (w-tx) so if
                                        we want to adjust
1003                // we have to apply w - (tx+gridX) and then
                                        do the adjustment
1004                tx = getWidth() - (gridDx *
(int)((getWidth() - (tx - gridDx))/gridDx));
1005                rp=true;
1006             }
1007
1008             if (e.getY() >= getHeight() - (dys/2)){
1009                 // Mouse is inside limits, move translation
                                             one cell up
1010                 // and adjust it to grid boundary.
1011                 // Same as before for adjustment.
1012                 ty = getHeight() - (gridDy *
                    (int)((getHeight() - (ty - gridDy))/gridDy));
1013                 rp=true;
```

```
1014            }
1015
1016            // We paint only if needed...
1017            if(rp){
1018                repaint();
1019            }
1020
1021            return;
1022        }
1023
1024        /*  [TFG] END  */
1025        tx += (e.getX() - mousePrevX)/scale;
1026        ty += (e.getY() - mousePrevY)/scale;
1027        mousePrevX = e.getX();
1028        mousePrevY = e.getY();
1029
1030        repaint();
1031    }
```

D. MODIFICACIONES EN TILEMAP

```
274     public void restore() {
275
276         height = resetMap.height;
277         width = resetMap.width;
278
279         imageKeys = new char[width][height];
280
281         for (int x = 0; x < width; x++) {
282             System.arraycopy(resetMap.imageKeys[x], 0,
                                 imageKeys[x], 0, height);
283         }
284
285         painted = new Robot.Paint[width][height];
286
287         for (int x = 0; x < width; x++) {
288             System.arraycopy(resetMap.painted[x], 0,
                                 painted[x], 0, height);
289         }
290
291         obstacleTiles = new Animation[width][height];
292         for (int x = 0; x < width; x++) {
293             System.arraycopy(resetMap.obstacleTiles[x], 0,
                                 obstacleTiles[x], 0, height);
294         }
295
296         decorationTiles = new Animation[width][height];
297         for (int x = 0; x < width; x++) {
298             System.arraycopy(resetMap.decorationTiles[x], 0,
                                 decorationTiles[x], 0, height);
299         }
300
301         strokes.clear();
302         for (Stroke2 s : resetMap.strokes) {
303             strokes.add(new Stroke2(s));
304         }
305
306         beacons.clear();
307         for (Beacon b : resetMap.beacons) {
308             beacons.add(new Beacon(b));
309         }
310
311         realAnimations = new
                    ArrayList<Animation>(resetMap.realAnimations);
312         robotStartX = resetMap.robotStartX;
313         robotStartY = resetMap.robotStartY;
314     }
```

```
462     public void updateStroke(int x, int y, Stroke2.Type t,
                                 Robot.Paint p) {
463         synchronized (strokes) {
464             /* [TFG] Original code:
465              *
466              * strokes.add(new Stroke2(x, y, t, p));
467              *
468              * Modified: in order to ommit duplicates, before
                   adding
469              * a new stroke check if that point is painted.
                 */
470
471             Stroke2 s = new Stroke2(x, y, t, p);
472
473             if (strokes.contains(s)
474                 || (t == Stroke2.Type.dot
475                     && strokes.contains(new Stroke2(x, y,
                                    Stroke2.Type.right, p)))
476                 || (t == Stroke2.Type.dot
477                     && strokes.contains(new Stroke2(x - 1, y,
                                    Stroke2.Type.right, p)))
478                 || (t == Stroke2.Type.dot
479                     && strokes.contains(new Stroke2(x, y,
                                     Stroke2.Type.down, p)))
480                 || (t == Stroke2.Type.dot
481                     && strokes.contains(new Stroke2(x, y - 1,
                                   Stroke2.Type.down, p)))) {
482                 System.err.println("Stroke ommited.");
483                 return;
484             }
485
486             Stroke2 tmp;
487
488             if (t == Stroke2.Type.right
489                 && strokes.contains(tmp = new Stroke2(x + 1,
                                y, Stroke2.Type.dot, p))) {
490                 strokes.remove(tmp);
491                 painted[x + 1][y] = null;
492                 System.err.println("Redundant right point
                                        stroke removed");
493             } else if (t == Stroke2.Type.down
494                     && strokes.contains(tmp = new
                       Stroke2(x, y + 1, Stroke2.Type.dot, p))) {
495                 strokes.remove(tmp);
496                 painted[x][y + 1] = null;
497                 System.err.println("Redundant down point
                                        stroke removed");
498             }
499
500             strokes.add(s);
501         }
502
```

```
503         if (t == Stroke2.Type.dot) {
504             painted[x][y] = p;
505         } else if (t == Stroke2.Type.down) {
506             painted[x][y] = painted[x][y + 1] = p;
507         } else if (t == Stroke2.Type.right) {
508             painted[x][y] = painted[x + 1][y] = p;
509         }
510 
511     }
```

```
735     public char[][] getImageKeys() {
736         return imageKeys;
737     }
738     private boolean modified = false;
739 
740     public void setModified(boolean modified) {
741         this.modified = modified;
742     }
743 
744     public boolean isModified() {
745         return modified;
746     }
747 
748 
749     public void extend(int w, int h) {
750         if (w < width && h < height) {
751             return;
752         }
753 
754         if (w < width) {
755             w = width;
756         } else {
757             w++;
758         }
759 
760         if (h < height) {
761             h = height;
762         } else {
763             h++;
764         }
765 
766         if (painted != null) {
767             painted = Arrays.copyOf(painted, w);
768 
769             for (int x = 0; x < w; x++) {
770                 if (painted[x] != null) {
771                     painted[x] = Arrays.copyOf(painted[x],
h);
772                 } else {
```

```
773                     painted[x] = new Robot.Paint[h];
774                 }
775             }
776         } else {
777             painted = new Robot.Paint[w][h];
778         }
779 
780         if (obstacleTiles != null) {
781             obstacleTiles = Arrays.copyOf(obstacleTiles, w);
782 
783             for (int x = 0; x < w; x++) {
784                 if (obstacleTiles[x] != null) {
785                     obstacleTiles[x] =
                                Arrays.copyOf(obstacleTiles[x], h);
786                 } else {
787                     obstacleTiles[x] = new Animation[h];
788                 }
789             }
790         } else {
791             obstacleTiles = new Animation[w][h];
792         }
793 
794         if (decorationTiles != null) {
795             decorationTiles = Arrays.copyOf(decorationTiles,
                                                              w);
796 
797             for (int x = 0; x < w; x++) {
798                 if (decorationTiles[x] != null) {
799                     decorationTiles[x] =
                            Arrays.copyOf(decorationTiles[x], h);
800                 } else {
801                     decorationTiles[x] = new Animation[h];
802                 }
803             }
804         }else{
805             decorationTiles = new Animation[w][h];
806         }
807 
808         if (imageKeys != null) {
809             imageKeys = Arrays.copyOf(imageKeys, w);
810 
811             for (int x = 0; x < w; x++) {
812                 if (imageKeys[x] != null) {
813                     imageKeys[x] =
                                    Arrays.copyOf(imageKeys[x], h);
814                 } else {
815                     imageKeys[x] = new char[h];
816                 }
817             }
818         }  else {
819             imageKeys = new char[w][h];
820         }
```

```
821
822         for (int x = width; x < w; x++) {
823             for (int y = 0; y < h; y++) {
824                 imageKeys[x][y] = ' ';
825             }
826         }
827
828         for (int y = height; y < h; y++) {
829             for (int x = 0; x < w; x++) {
830                 imageKeys[x][y] = ' ';
831             }
832         }
833
834
835         width = w;
836         height = h;
837     }
838
839     public void extendBeginning(int w, int h) {
840         if (w < width && h < height) {
841             return;
842         }
843
844         if (w < width) {
845             w = width;
846         }
847
848         if (h < height) {
849             h = height;
850         }
851
852         int iw = w - width;
853         int ih = h - height;
854
855         Robot.Paint[][] oldPainted = painted;
856         painted = new Robot.Paint[w][h];
857
858         if (oldPainted != null) {
859             for (int x = 0; x < width; x++) {
860                 if (oldPainted[x] != null) {
861                     System.arraycopy(oldPainted[x], 0,
                                      painted[x + iw], ih, height);
862                 }
863             }
864         }
865
866
867         Animation[][] oldObstacleTiles = obstacleTiles;
868         obstacleTiles = new Animation[w][h];
869
870         if (oldObstacleTiles != null) {
871             for (int x = 0; x < width; x++) {
```

```
872                 if (oldObstacleTiles[x]!=null){
873                     System.arraycopy(oldObstacleTiles[x], 0,
                                obstacleTiles[x + iw], ih, height);
874                 }
875             }
876         }
877 
878         Animation[][] oldDecorationTiles = decorationTiles;
879         decorationTiles = new Animation[w][h];
880 
881         if (oldDecorationTiles != null){
882             for (int x = 0; x < width; x++) {
883                 if (oldDecorationTiles[x] != null) {
884                     System.arraycopy(oldDecorationTiles[x],
                             0, decorationTiles[x + iw], ih, height);
885                 }
886             }
887         }
888 
889         char[][] oldImageKeys = imageKeys;
890 
891         imageKeys = new char[w][h];
892 
893         if (oldImageKeys != null){
894             for (int x = 0; x < width; x++) {
895                 if (oldImageKeys[x] != null) {
896                     System.arraycopy(oldImageKeys[x], 0,
                                imageKeys[x + iw], ih, height);
897                 }
898             }
899         }
900 
901         for (int x = 0; x < iw; x++) {
902             for (int y = 0; y < h; y++) {
903                 imageKeys[x][y] = ' ';
904             }
905         }
906 
907         for (int y = 0; y < ih; y++) {
908             for (int x = 0; x < w; x++) {
909                 imageKeys[x][y] = ' ';
910             }
911         }
912 
913         for (Stroke2 s2 : getStrokes()) {
914             s2.x += iw;
915             s2.y += ih;
916         }
917 
918         for (Beacon b : getBeacons()) {
919             b.x += iw;
920             b.y += ih;
```

```
921         }
922
923         robotStartX += iw;
924         robotStartY += ih;
925
926         width = w;
927         height = h;
928     }
```

BIBLIOGRAFÍA

[1] Página principal de RoboMind
http://robomind.net
[2] Robo Programming Structures
http://www.robomind.net/en/docProgrammingStructures.htm
[3] Lego Mindstorms NXT Compatibility
http://www.robomind.net/en/docLego.html
[4] Página principal de Alice
http://www.alice.org/index.php
[5] ¿Qué es Alice?
http://www.alice.org/index.php?page=what_is_alice/what_is_alice
[6] Creating from Scratch
http://web.mit.edu/newsoffice/2007/resnick-scratch.html
[7] Página principal de Scractch
http://scratch.mit.edu/
[8] Skratch Getting Started:
http://info.scratch.mit.edu/sites/infoscratch.media.mit.edu/files/file/ScratchGettingStartedv14.pdf
[9] Patrón MVC
http://www.fdi.ucm.es/profesor/jpavon/poo/2.14.mvc.pdf
[10] Manual creación aplicación Java MVC
http://www.re-orientation.com/manual-creacion-aplicacion-java-MVC
[11] Definición básica MVC
http://www.lab.inf.uc3m.es/~a0080802/RAI/mvc.html
[12] Singleton Pattern
http://www.oodesign.com/singleton-pattern.html
[13] Java SE Application Design With MVC
http://www.oracle.com/technetwork/articles/javase/mvc-136693.html
[14] AST intermediate representations
http://lambda-the-ultimate.org/node/716
[15] A Tree-Based Alternative to Java Byte-Codes
http://www.ics.uci.edu/~franz/Site/pubs-pdf/C05.pdf
[16] Tree vs Byte
http://central.kaserver5.org/Kasoft/Typeset/JavaTree/index.html
[17] Flamingo tutorial
http://blog.frankel.ch/flamingo-tutorial
[18] Google Blocky
https://code.google.com/p/blockly/
[19] Abstract Syntaxt Tree
http://en.wikipedia.org/wiki/Abstract_syntax_tree

[20] Syntax Tree Generator
http://mshang.ca/syntree/

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Feb 14 20:08:30 CET 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)